IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Appl. No.:	10/621,067	§	Confirmation No.:	1252
Applicant:	Keith Farkas	§		
Filed:	07/16/2003	§		
TC/A.U.:	2195	§		
Examiner:	Kenneth Tang	§		
Title:	HETEROGENEOUS PROCESSOR CORE SYSTEMS FOR IMPROVED THROUGHPUT	§		
Docket No.:	200210109-1 (HPC.0762US)	§		

Mail Stop Appeal Brief-Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF PURSUANT TO 37 C.F.R § 41.37

Sir:

The final rejection of claims 1-2, 7-8, 15, 18-20, 22-26 and 29-37 is hereby appealed.

I. REAL PARTY IN INTEREST

The real party in interest is the Hewlett-Packard Development Company, LP. The Hewlett-Packard Development Company, LP, a limited partnership established under the laws of the State of Texas and having a principal place of business at 11445 Compaq Center Drive West, Houston, TX 77070, U.S.A. (hereinafter "HPDC"). HPDC is a Texas limited partnership and is a wholly-owned affiliate of Hewlett-Packard Company, a Delaware Corporation, headquartered in Palo Alto, CA. The general or managing partner of HPDC is HPQ Holdings, LLC.

II. RELATED APPEALS AND INTERFERENCES

None.

III. STATUS OF THE CLAIMS

Claims 1-2, 7-8, 15, 18-20, 22-26 and 29-37 have been finally rejected and are the subject of this appeal.

Claims 3-6, 9-14, 16, 17, 21 and 27-28 have been cancelled.

IV. STATUS OF AMENDMENTS

No amendment after the final rejection of August 4, 2010 has been submitted. Therefore, all amendments have been entered.

V. SUMMARY OF THE CLAIMED SUBJECT MATTER

The following provides a concise explanation of the subject matter defined in each of the independent claims involved in the appeal, referring to the specification by page and line number and to the drawings by reference characters, as required by 37 C.F.R. § 41.37(c)(1)(v). Each element of the claims is identified by a corresponding reference to the specification and drawings where applicable. Note that the citation to passages in the specification and drawings for each claim element does not imply that the limitations from the specification and drawings should be read into the corresponding claim element. Note also that the cited passages are provided as examples, as other passages in the specification or drawings not cited may also be relevant to the corresponding claim elements.

Independent claim 1 recites a computer system, comprising:

a plurality of computer processor cores (Fig. 1:101-107) in which at least two of the computer processor cores are heterogeneous, and wherein the plurality of computer processor cores are configured to execute the same instruction set (Spec., p. 4, ln. 1-30); and

a performance measurement and transfer mechanism (Fig. 1:108) configured to move a plurality of executing computer processing jobs amongst the plurality of computer processor cores by matching requirements of the plurality of executing computer processing jobs to processing capabilities of the computer processor cores (Spec., p. 4, ln. 31-33; p. 12, ln. 2-6; p. 14, ln. 8-12).

Independent claim 7 recites a method for operating multiple processor cores, comprising:

obtaining a throughput metric that identifies throughput achieved by a plurality of computer processor cores (Fig. 1:101-107) as a function of workloads running on said computer processor cores, wherein the plurality of computer processor cores are on a single semiconductor die (Spec., p. 5, ln. 20-26; p. 11, ln. 14-16; p. 15, ln. 7-9), in which at least two computer processor cores differ in processing capability, and wherein the computer processor cores execute the same instruction set (Spec., p. 4, ln. 1-5); and

transferring (Fig. 2:208, 216) individual ones of a plurality of computer processing jobs amongst targeted ones of said plurality of computer processor cores based on the throughput metric (Spec., p. 8, ln. 29 – p. 9, ln. 25; p. 11, ln. 11-34; p. 12, ln. 17-22; p. 15, ln. 10-14).

Independent claim 20 recites a computer system, comprising:

a plurality of computer processor cores (Fig. 1:101-107) in which at least two differ in processing performance, and wherein the plurality of computer processor cores are configured to execute the same instruction set (Spec., p. 4, ln. 1-30); and

a performance measurement and transfer mechanism (Fig. 1:108) configured to move a plurality of executing computer processing jobs amongst the plurality of computer processor cores based on a measured throughput metric (Spec., p. 4, ln. 31-33; p. 12, ln. 2-6; p. 14, ln. 8-12),

wherein the performance measurement and transfer mechanism is configured to swap execution of the executing computer processing jobs between the computer processor cores for a period of time, monitor resulting performance, and then build a data structure with relative performances of jobs on different types of the computer processor cores (Spec., p. 9, ln. 11-25).

Independent claim 25 recites a method for operating multiple processor cores, comprising:

assigning a plurality of computer processing jobs amongst a plurality of computer processor cores (Fig. 1:101-107) , wherein at least two of the computer processor cores differ in size or complexity but execute the same instruction set (Spec., p. 4, ln. 1-30; p. 5, ln. 26-30), and

wherein assigning the plurality of computer processing jobs amongst the plurality of computer processor cores comprises matching requirements of the computer processing jobs to processing capabilities of the computer processor cores based on the sizes or complexities of the computer processor cores (Spec., p. 4, ln. 31-33; p. 12, ln. 2-6; p. 14, ln. 8-12).

Independent claim 29 recites a method for operating multiple processor cores, comprising:

obtaining a throughput metric that identifies throughput achieved by computer processor cores (Fig. 1:101-107) on a single semiconductor die as a function of workloads running on said computer processor cores (Spec., p. 5, ln. 20-36; p. 11, ln. 14-16; p. 15, ln. 7-9); and

assigning a plurality of computer processing jobs amongst the computer processor cores (Fig. 1:101-107) based on the throughput metric, wherein at least two of the computer processor cores differ in size or complexity but execute the same instruction set (Spec., p. 4, ln. 1-5; p. 8, ln. 29 – p. 9, ln. 25; p. 11, ln. 11-34);

transferring (Fig. 2:208, 216) the computer processing jobs to a new assignment amongst the computer processor cores (Spec., p. 8, ln. 29 – p. 9, ln. 25; p. 11, ln. 11-34; p. 12, ln. 17-22);

collecting (Fig. 2:202, 210) statistics about execution performance of the computer processing jobs at the new assignment (Spec., p. 10, ln. 33-36; p. 11, ln. 14-16);

determining (Fig. 2:214) whether to reassign the computer processing jobs to different computer processor cores based on the statistics collected (Spec., p. 11, ln. 19-21); and

building a data structure with relative performances of the computer processing jobs on different types of computer processor cores based on the statistics collected (Spec., p. 9, ln. 21-25).

VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

A. Claims 1-2, 19, 22, 32, and 34 were rejected under 35 U.S.C. § 102(a) as anticipated by the Kumar Abstract ("Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures").

B. Claims 7-8, 15, 18, 24, 29-30, and 37 were rejected under 35 U.S.C. § 103(a) as unpatentable over the Kumar Abstract in view of Calder (U.S. Patent Publication No. 2004/0111708).

C. Claims 20 and 33 were rejected under 35 U.S.C. § 103(a) as unpatentable over the Kumar Abstract.

D. Claims 23 and 31 were rejected as unpatentable over the Kumar Abstract in view of Calder and further in view of Nagae (U.S. Patent No. 6,006,248).

E. Claims 25-26 and 35-36 were rejected under 35 U.S.C. § 103(a) as unpatentable over the Kumar Abstract in view of Paker ("A heterogeneous multi-core platform for low power signal processing in systems-on-chip").

F. Claims 1-2, 7-8, 15, 18-19, 22, 24, and 32-34 were rejected under 35 U.S.C. § 102(a) as anticipated by Calder.

G. Claims 20-21, 29-30, and 37 were rejected under 35 U.S.C. § 103(a) as unpatentable over Calder.

H. Claims 23 and 31 were rejected under 35 U.S.C. § 103(a) as unpatentable over Calder in view of Nagae.

I. Claims 25-26 and 35-36 were rejected under 35 U.S.C. § 103(a) as unpatentable over Calder in view of Paker.

VII. ARGUMENT

The claims do not stand or fall together. Instead, Appellant presents separate arguments for various independent and dependent claims. Each of these arguments is separately argued below and presented with separate headings and sub-headings as required by 37 C.F.R. § 41.37(c)(1)(vii).

A. Claims 1-2, 19, 22, 32, and 34 were rejected under 35 U.S.C. § 102(a) as anticipated by the Kumar Abstract ("Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures").

1. Claims 1, 32, 34.

Independent claim 1 and its dependent claims were rejected as purportedly anticipated by the Kumar Abstract, which is printout of a web page from portal.acm.org containing an Abstract of a paper by Rakesh Kumar et al. The Kumar Abstract purports to have a publication date of January 2002.

As established by the Second Declaration under 37 C.F.R. § 1.132, submitted with the Amendment of May 10, 2010, it is clear that the 2002 publication date for the Kumar Abstract is in error. *See* Second Declaration under 37 C.F.R. § 1.132, ¶¶ 14-17.

As established by the Second Declaration under 37 C.F.R. § 1.132, it would not be possible for the Kumar Abstract to have been published in 2002, since the corresponding paper by Rakesh Kumar et al., entitled "Processor Power Reduction by a Single-ISA Heterogeneous Multi-Core Architectures," was not submitted to the publication entity of the IEEE Computer Architecture Letters until March 2003.[1] *Id.*, ¶¶ 14, 17. Moreover, the inventors also state in the Second Declaration under 37 C.F.R. § 1.132 that any abstract relating to the above-referenced paper was not submitted to any publication entity for publication in 2002, and thus, the Kumar Abstract could not have been published in 2002.

The Response to Arguments section of the final Office Action argued that the Second Declaration under 37 C.F.R. § 1.132 is insufficient to overcome the rejection because "the evidence is not found to be persuasive because it does not rely on facts, but instead, relies on

[1] Note that this Kumar paper does not constitute prior art against the present invention in view of the First Declaration Under 37 C.F.R. § 1.132 submitted with the Amendment of October 2, 2009.

mere recollection that the first submission of the paper ... was believed to be been submitted in March 2003 ...." 08/04/2010 Office Action at 24.

This assertion is erroneous. The Second Declaration under 37 C.F.R. § 1.132 clearly sets forth actual evidence that corroborates the inventors' recollection of the events surrounding the publication of the Kumar paper and the incorrect date associated with the Kumar Abstract. Exhibit C to the Second Declaration under 37 C.F.R. § 1.132 depicts the Table of Contents for the 2002 publication of the IEEE Computer Architecture Letters, where this Table of Contents does **not** contain any listing for Rakesh Kumar et al., "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures." Second Declaration under 37 C.F.R. § 1.132, ¶ 15. On the other hand, Exhibit D to the Second Declaration under 37 C.F.R. § 1.132 shows the Table of Contents for the 2003 publication of IEEE Computer Architecture Letters, which includes a listing for the foregoing Kumar paper. *Id.*, ¶ 16. Exhibits C and D contain information from the publisher of the Kumar paper, and such information clearly indicates that the Kumar paper was published in the 2003 publication of the IEEE Computer Architecture Letters, **not** in the 2002 publication of the IEEE Computer Architecture Letters. *Id.*, ¶ 17. Such evidence clearly corroborates the inventor's recollection that the Kumar Abstract was not submitted to the publisher until March 2003 or later. *Id.*, ¶¶ 14, 17.

In view of the foregoing, it is clearly established that the Kumar Abstract does not have a publication date of 2002, but rather, has a publication date of 2003.

Since the "2002" publication date attributed to the Kumar Abstract is in error, the Kumar Abstract does not constitute prior art under 35 U.S.C. § 102(b) against the present application. Moreover, since the Kumar Abstract contains content of the above-referenced paper, as stated in

¶ 14 of the Second Declaration under 37 C.F.R. § 1.132, the Kumar Abstract does not constitute prior art under any section of 35 U.S.C. § 102.

In view of the foregoing, since the Kumar Abstract does not constitute prior art under § 102, the § 102 rejection of the foregoing claims is clearly erroneous.

Reversal of the final rejection of the above claims is respectfully requested.

2. Claim 2.

Claim 2 depends from claim 1 and is therefore allowable for at least the same reasons as claim 1. Moreover, claim 2 further recites that the at least one of the operating system, firmware, and hardware is configured to provide for a **periodic test** to determine **relative performance** of different computer processing jobs on different ones of the computer processor cores.

The Kumar Abstract clearly does not provide any teaching of a periodic test to determine **relative performance** of different computer processing jobs on different ones of the computer processor cores. The Kumar Abstract states that during an application's execution, system software dynamically chooses the most appropriate core to meet specific performance and power **requirements**. This does not refer to a **periodic test**, and this does not refer to determining relative performance of different computer processing jobs on different ones of the computer processor cores. The Kumar Abstract merely refers to matching based on specific **requirements**, which include performance and power **requirements**.

Claim 2 is therefore further allowable for the foregoing reasons.

Reversal of the final rejection of the above claim is respectfully requested.

3. Claims 19, 22.

Claim 19 depends from claim 1 and is therefore allowable for at least the same reasons as claim 1. Moreover, claim 19 further recites:

> wherein the performance measurement and transfer mechanism is configured to transfer the executing computer processing jobs to a new assignment amongst the plurality of computer processor cores, collect performance statistics about execution at the new assignment, and then determine whether to reassign the executing computer processing jobs to different computer processor cores based on the performance statistics collected.

There is nothing in the Kumar Abstract that relates to collecting performance **statistics** about execution at the new assignment, and then determining whether to reassign the executing computer processing jobs to different computer processor cores based on the performance **statistics** collected. The Kumar Abstract refers to choosing the most appropriate core to meet specific **requirements**, including performance and power **requirements**. There is no hint of collecting performance **statistics** about execution at the new assignment, and then determining whether to reassign based on the collected performance **statistics**.

Claim 19 and its dependent claim 22 are therefore further allowable for the foregoing reasons.

Reversal of the final rejection of the above claims is respectfully requested.

B. Claims 7-8, 15, 18, 24, 29-30, and 37 were rejected under 35 U.S.C. § 103(a) as unpatentable over the Kumar Abstract in view of Calder (U.S. Patent Publication No. 2004/0111708).

1. Claims 7, 15, 18.

The obviousness rejection of independent claim 7 is also erroneous.

First, the rejection is defective since the Kumar Abstract does not constitute prior art against the present invention, as explained above in connection with claim 1.

Second, the rejection is based on the incorrect assertion by the Examiner that the Kumar Abstract discloses "obtaining a metric achieved by a plurality of computer processor cores as a function of workloads running on said computer processor cores." 08/04/2010 Office Action at 5. The Kumar Abstract specifically discloses dynamically choosing the most appropriate core to meet specific requirements, including performance and power requirements. However, choosing a core based on specific requirements has nothing to do with obtaining a **metric achieved** by a plurality of computer processor cores as a function of workloads running on the computer processor cores. The term "metric" that is "achieved" by computer processor cores as a function of workloads running on the computer processor cores indicates some type of a statistic relating to execution of the workloads on the processor cores. However, the Kumar Abstract merely illustrates the choosing of a core that meets specific **requirements**.

Therefore, it is clear that, contrary to the allegation by the Examiner, the Kumar Abstract does not disclose or hint at the foregoing claimed subject matter.

Moreover, the allegation that that Kumar Abstract discloses the first element of claim 7 is based on a rewriting of this claim element, in which the term "throughput" was deleted. The actual language of the "obtaining" clause of claim 7 is "obtaining a **throughput** metric that identifies **throughput** achieved by a plurality of computer processor cores as a function of workloads running on said computer processor cores." As conceded by the Examiner, the Kumar Abstract does not disclose such a throughput metric. *Id.*

Instead, the Examiner cited Calder as purportedly disclosing the tracking of statistics. *Id.* at 6. One of the passages relied upon by the Examiner is ¶ [0049], lines 1-4, of Calder. *Id.*

Note that Calder has a filing date of September 9, 2003, which is **<u>after</u>** the filing date of the present application (July 16, 2003).

Calder claims the benefit of the following two provisional applications, each having a filing date of September 9, 2002: provisional applications Serial Nos. 60/409,105 (hereinafter "'105 Provisional Application"), and 60/409,106 (hereinafter "'106 Provisional Application").

However, the '105 Provisional Application and the '106 Provisional Application do not contain content of Calder relied upon by the Office Action in rendering all rejections based on Calder. Specifically, the rejection relied upon ¶ [0049] of Calder, which refers to a single ISA architecture that has several heterogeneous cores. Paragraph [0049] of Calder also states that the resource requirements of the program during execution are determined on a per phase basis. Paragraph [0049] of Calder also states that the resource requirements determine which of the multi-core architectures a given phase should run on. According to ¶ [0049] of Calder, the phase classification guides each phase of the program execution to a specific core.

The content of ¶ [0049] of Calder is not supported by either the '105 Provisional Application or '106 Provisional Application. Copies of both the '105 and '106 Provisional Applications were attached to the Amendment dated May 10, 2010. A thorough review of the '105 Provisional Application indicates that there is no reference in the '105 Provisional Application to any one of the following concepts: heterogeneous cores, multi-core architecture, or guiding each phase of program execution to a specific core.

The '106 Provisional Application also does not provide any discussion of the content of ¶ [0049] of Calder. The '106 Provisional Application refers to a technique for tracking a metric to classify phases of a program, and to predict program phases. '106 Provisional Application, pages 1-2. As stated on page 5 of the '106 Provisional Application, the content of the '106 Provisional Application includes a "unified phase tracking architecture" and a "phase change prediction architecture." Section 5 on pages 10-18 of the '106 Provisional Application describe an architecture for capturing phases, such that phases of a program can be classified. Section 6 of the '106 Provisional Application on pages 19-21 describe predicting a next phase that is to occur.

Section 5.4 on page 15 of the '106 Provisional Application states that "phase classifications on programs at run-time [have] little to no impact on the design of the processor core."

However, nowhere in the '106 Provisional Application is there any discussion regarding a phase classification that guides each phase of the program execution to a specific core of a multi-core architecture that has several heterogeneous cores, as stated in ¶ [0049] of Calder.

Therefore, at least ¶ [0049] of Calder clearly constitutes material added to Calder that did not exist in either the '105 or '106 Provisional Applications. Therefore, at least ¶ [0049] of Calder has a priority date of September 9, 2003, which is **after** the filing date of the present application.

Thus, at least ¶ [0049] of Calder does not constitute prior art against the present application.

In view of the foregoing, it is clear that Calder would not have provided any hint of obtaining a throughput metric that identifies throughput achieved by computer processor cores as a function of workloads running on the computer processor cores, where the plurality of computer processor cores differ in processing capability.

The other passages, ¶ [0055] and claim 51, of Calder relied upon by the Examiner also do not provide any hint of the foregoing throughput metric.

Therefore, even if the Kumar Abstract and Calder could be hypothetically combined, the hypothetical combination of references would not have led to the claimed subject matter. The obviousness rejection of claim 7 and its dependent claims is therefore erroneous.

Reversal of the final rejection of the above claims is respectfully requested.

2. Claim 8.

Claim 8 depends from claim 7 and is therefore allowable for at least the same reasons as claim 7. Moreover, claim 8 further recites:

> providing for a periodic test to determine relative performance of different computer processing jobs on different ones of the computer processor cores.

As discussed above in connection with claim 2, and contrary to the allegation made by the Examiner on page 6 of the final Office Action, the Kumar Abstract clearly does not provide any hint of providing a periodic **test** to determine **relative performance** of different computer processing jobs on different ones of the computer processor cores. Claim 8 is therefore further allowable for the foregoing reasons.

Reversal of the final rejection of the above claims is respectfully requested.

3. Claim 24.

In view of the allowability of base claim 1 over the Kumar Abstract, the obviousness rejection of claim 24 over the Kumar Abstract and Calder has been overcome.

Reversal of the final rejection of the above claim is respectfully requested.

4. Claims 29, 30, 37.

The obviousness rejection of independent claim 29 is also erroneous.

For reasons stated above with respect to claim 1, the Kumar Abstract does not constitute prior art with respect to the present invention. Moreover, as explained above in connection with claim 7, a passage (¶ [0049]) of Calder relied upon by the Examiner also does not constitute prior art against the present invention.

Also, for reasons similar to those stated above with respect to claim 7, it is clear that the hypothetical combination of the Kumar Abstract and Calder fail to disclose at least the following elements of claim 29:

> obtaining a throughput metric that identifies throughput achieved by computer processor cores on a single semiconductor die as a function of workloads running on said computer processor cores; and
>
> assigning a plurality of computer processing jobs amongst the computer processor cores based on the throughput metric, wherein at least two of the computer processor cores differ in size or complexity but execute the same instruction set.

Moreover, the Examiner erred in arguing that the Kumar Abstract discloses "collecting statistics about execution performance of the computer processing jobs at the new assignment." 08/04/2010 Office Action at 8. The Kumar Abstract refers to choosing the most appropriate core to meet specific **requirements**. There is absolutely no hint given in the Kumar Abstract regarding collecting **statistics** about execution performance of the computer processing jobs.

Moreover, the Examiner conceded that both the Kumar Abstract and Calder fail to disclose the following clause of claim 29:

> building a data structure with relative performances of the computer processing jobs on different types of computer processor cores based on the statistics collected.

08/04/2010 Office Action at 9. Despite this concession, the Examiner argued that it would have been obvious to build such a data structure. No objective evidence was cited to support this argument. It is clear that neither the Kumar Abstract nor Calder provides any hint of building such a data structure with relative performances of the computer processing jobs on different types of computer processor cores based on the statistics collected.

Therefore, in view of the foregoing, it is clear that the obviousness rejection of claim 29 and its dependent claims is erroneous.

Reversal of the final rejection of the above claims is respectfully requested.

C. Claims 20 and 33 were rejected under 35 U.S.C. § 103(a) as unpatentable over the Kumar Abstract.

1. Claim 20.

Independent claim 20 was rejected as purportedly obvious over the Kumar Abstract alone. In view of the fact that the Kumar Abstract does not constitute prior art against the present invention, the rejection of claim 20 is clearly in error.

In the rejection, the Examiner conceded that Kumar does not disclose building a data structure with relative performances of jobs on different types of the computer processor cores. 08/04/2010 Office Action at 10.

However, in view of the discussion with respect to claim 29, this allegation is clearly defective, since no objective evidence was cited by the Examiner to support this allegation.

Moreover, the rejection is defective based on the incorrect allegation that the Kumar Abstract discloses a performance measurement and transfer mechanism that is configured to move a plurality of executing computer processing jobs amongst computer processor cores based on a measured throughput metric. The Kumar Abstract discloses choosing the most appropriate core given specific **requirements**—there is no hint here of moving computer processing jobs based on a measured throughput metric.

In view of the foregoing, it is clear that the obviousness rejection of claim 20 is erroneous.

Reversal of the final rejection of the above claims is respectfully requested.

2. Claim 33.

In view of the allowability of base claim 1 over the Kumar Abstract, the obviousness rejection of dependent claim 33 is also erroneous.

Reversal of the final rejection of the above claim is respectfully requested.

D. Claims 23 and 31 were rejected as unpatentable over the Kumar Abstract in view of Calder and further in view of Nagae (U.S. Patent No. 6,006,248).

1. Claims 23, 31.

In view of the allowability of base claims 7 and 29 over the Kumar Abstract in view of Calder, the obviousness rejection of dependent claims 23 and 31 over the Kumar Abstract, Calder and Nagae has been overcome.

Reversal of the final rejection of the above claims is respectfully requested.

E. Claims 25-26 and 35-36 were rejected under 35 U.S.C. § 103(a) as unpatentable over the Kumar Abstract in view of Paker ("A heterogeneous multi-core platform for low power signal processing in systems-on-chip").

1. Claims 25, 35, 36.

The obviousness rejection of independent claim 25 over the Kumar Abstract and Paker is erroneous for the reason that the Kumar Abstract does not constitute prior art with respect to the present application. Therefore, the obviousness rejection of claim 25 and its dependent claims over the Kumar Abstract and Paker is clearly erroneous.

Reversal of the final rejection of the above claims is respectfully requested.

2. Claim 26.

Claim 26 depends from claim 25 and is therefore allowable for at least the same reasons as claim 25. Moreover, claim 26 is further allowable based on the incorrect allegation by the Office Action that the Kumar Abstract discloses periodically testing to determine relative performance of different computer processing jobs on different ones of the computer processor cores. 08/04/2010 Office Action at 13. As discussed above in connection with claim 2, it is clear that the Kumar Abstract does not provide any hint of the foregoing subject matter. Claim 26 is therefore further allowable for the foregoing reasons.

Reversal of the final rejection of the above claim is respectfully requested.

F. Claims 1-2, 7-8, 15, 18-19, 22, 24, and 32-34 were rejected under 35 U.S.C. § 102(a) as anticipated by Calder.

1. Claims 1, 2, 24, 32-34.

It is respectfully submitted that the § 102 rejection of claim 1 over Calder is erroneous.

Note that Calder has a filing date of September 9, 2003, which is **after** the filing date of the present application (July 16, 2003).

Calder claims the benefit of the following two provisional applications, each having a filing date of September 9, 2002: provisional applications Serial Nos. 60/409,105 (hereinafter "'105 Provisional Application"), and 60/409,106 (hereinafter "'106 Provisional Application").

However, the '105 Provisional Application and the '106 Provisional Application do not contain content of Calder relied upon by the Office Action in rendering all rejections based on Calder. Specifically, the rejection relied upon ¶ [0049] of Calder, which refers to a single ISA architecture that has several heterogeneous cores. Paragraph [0049] of Calder also states that the resource requirements of the program during execution are determined on a per phase basis.

Paragraph [0049] of Calder also states that the resource requirements determine which of the multi-core architectures a given phase should run on. According to ¶ [0049] of Calder, the phase classification guides each phase of the program execution to a specific core.

The content of ¶ [0049] of Calder is not supported by either the ‘105 Provisional Application or ‘106 Provisional Application. Copies of both the ‘105 and ‘106 Provisional Applications were attached to the Amendment dated May 10, 2010. A thorough review of the ‘105 Provisional Application indicates that there is no reference in the ‘105 Provisional Application to any one of the following concepts: heterogeneous cores, multi-core architecture, or guiding each phase of program execution to a specific core.

The ‘106 Provisional Application also does not provide any discussion of the content of ¶ [0049] of Calder. The ‘106 Provisional Application refers to a technique for tracking a metric to classify phases of a program, and to predict program phases. ‘106 Provisional Application, pages 1-2. As stated on page 5 of the ‘106 Provisional Application, the content of the ‘106 Provisional Application includes a “unified phase tracking architecture” and a “phase change prediction architecture.” Section 5 on pages 10-18 of the ‘106 Provisional Application describe an architecture for capturing phases, such that phases of a program can be classified. Section 6 of the ‘106 Provisional Application on pages 19-21 describe predicting a next phase that is to occur.

Section 5.4 on page 15 of the ‘106 Provisional Application states that “phase classifications on programs at run-time [have] little to no impact on the design of the processor core.”

However, nowhere in the '106 Provisional Application is there any discussion regarding a phase classification that guides each phase of the program execution to a specific core of a multi-core architecture that has several heterogeneous cores, as stated in ¶ [0049] of Calder.

Therefore, at least ¶ [0049] of Calder clearly constitutes material added to Calder that did not exist in either the '105 or '106 Provisional Applications. Therefore, at least ¶ [0049] of Calder has a priority date of September 9, 2003, which is **after** the filing date of the present application.

Thus, at least ¶ [0049] of Calder does not constitute prior art against the present application.

In view of the foregoing, withdrawal of the § 102 rejection of the claims over Calder is respectfully requested, since the rejection is based on content of Calder not entitled to the September 9, 2002 filing date of the '105 and '106 Provisional Applications.

The other passage of Calder relied upon by the Office Action in the rejection of claim 1 is the Abstract of Calder. The Abstract of Calder refers to tracking a statistic for a component, and using the tracked statistic to identify a behavior of a program over each of multiple intervals of execution. The Abstract further notes that the behavior of the interval is compared to the behavior of another interval of execution to find similar sections of behavior. Calder, Abstract. There is nothing in the Abstract of Calder that provides any hint of heterogeneous computer processor cores that are configured to execute the same instruction set, or a performance measurement and transfer mechanism configured to move executing computer processing jobs amongst the plurality of computer processor cores by matching requirements of the plurality of executing computer processing jobs to processing capabilities of the computer processor cores.

Claim 1 and its dependent claims are therefore clearly allowable over Calder.

Reversal of the final rejection of the above claims is respectfully requested.

2. Claims 7, 8, 15, 18, 19, 22.

The rejection of independent claim 7 over Calder also relies upon ¶ [0049] of Calder, which finds no support whatsoever in the '105 and '106 Provisional Applications from which Calder claims priority. Therefore, since ¶ [0049] does not have a priority date that pre-dates the filing date of the present application, the rejection of claim 7 over Calder is also erroneous.

The other passages of Calder relied upon by the Office Action against claim 7 include ¶ [0055], claim 51, and the Abstract of Calder. The Abstract of Calder provides no hint whatsoever of transferring individual ones of a plurality of computer processing jobs amongst targeted ones of the plurality of computer processor cores based on a throughput metric, where the plurality of computer processor cores are on a single semiconductor die, and where at least two computer processor cores differ in processing capability.

Claim 51 of Calder refers to using a statistic to perform one of a behavior optimization, statistic optimization, load-time optimization, run-time optimization, and hardware reconfiguration. However, there is no hint here of the "transferring" element of claim 7, in the context of computer processor cores at least two of which differ in processing capability and are on a single semiconductor die.

Paragraph [0055] of Calder describes tracking a statistic for components, where the statistic is a hardware metric and/or hardware-independent metric. However, no mention is made in ¶ [0055] of performing the transferring recited in claim 7.

In view of the foregoing, it is clear that claim 7 and its dependent claims are also allowable over Calder.

Reversal of the final rejection of the above claims is respectfully requested.

G. Claims 20-21, 29-30, and 37 were rejected under 35 U.S.C. § 103(a) as unpatentable over Calder.

1. Claim 20.

Independent claim 20 was rejected as purportedly obvious over Calder alone. In view of the Examiner's reliance on ¶ [0049] of Calder, which constitutes material added to Calder that did not exist in either the '105 or '106 Provisional Applications, it is respectfully submitted that the rejection is in error since the rejection is relying on subject matter of Calder that has a priority date of September 9, 2003, which is **after** the filing date of the present application.

The rejection of claim 20 is also premised on the allegation that building of a data structure with relative performances of jobs on different types of the computer processor cores would have been obvious, even though the Examiner conceded that Calder does not disclose this claimed feature. 08/04/2010 Office Action at 19. As discussed above in connection with claim 29, such allegation is clearly incorrect, as the Examiner has not cited to any objective evidence to support the allegation.

Therefore, the obviousness rejection of claim 20 is clearly erroneous.

Reversal of the final rejection of the above claim is respectfully requested.

2. Claims 29, 30, 37.

The obviousness rejection of independent claim 29 is also defective based on the reliance on ¶ [0049] of Calder, which does not constitute prior art with respect to the present application. Moreover, the rejection of claim 29 is also defective based on the unsupported allegation that building a data structure as recited in claim 29 would have been obvious.

Therefore, the obviousness rejection of claim 29 and its dependent claims is clearly erroneous.

Reversal of the final rejection of the above claims is respectfully requested.

H. Claims 23 and 31 were rejected under 35 U.S.C. § 103(a) as unpatentable over Calder in view of Nagae.

1. Claims 23, 31.

In view of the allowability of base claims 7 and 29 over Calder, the obviousness rejection of dependent claims 23 and 31 over Calder and Nagae has been overcome.

Reversal of the final rejection of the above claims is respectfully requested.

I. Claims 25-26 and 35-36 were rejected under 35 U.S.C. § 103(a) as unpatentable over Calder in view of Paker.

1. Claims 25, 26, 35, 36.

The obviousness rejection of independent claim 25 over Calder in view of Paker is erroneous based on the fact that the obviousness rejection relied upon ¶ [0049] of Calder, which does not constitute prior art with respect to the claimed subject matter. Therefore, the obviousness rejections of claim 25 and its dependent claims is clearly defective.

Reversal of the final rejection of the above claims is respectfully requested.

CONCLUSION

In view of the foregoing, reversal of all final rejections and allowance of all pending claims is respectfully requested.

Respectfully submitted,

Date: January 4, 2011

/Dan C. Hu/

Dan C. Hu
Registration No. 40,025
TROP, PRUNER & HU, P.C.
1616 South Voss Road, Suite 750
Houston, TX 77057-2631
Telephone: (713) 468-8880
Facsimile: (713) 468-8883

VIII. APPENDIX OF APPEALED CLAIMS

Claims 3-6, 9-14, 16, 17, 21 and 27-28 have been cancelled.

The claims on appeal are:

1. A computer system, comprising:

a plurality of computer processor cores in which at least two of the computer processor cores are heterogeneous, and wherein the plurality of computer processor cores are configured to execute the same instruction set; and

a performance measurement and transfer mechanism configured to move a plurality of executing computer processing jobs amongst the plurality of computer processor cores by matching requirements of the plurality of executing computer processing jobs to processing capabilities of the computer processor cores.

2. The computer system of claim 1, further comprising:

at least one of an operating system hosted on the plurality of computer processor cores, firmware, and hardware that includes the performance measurement and transfer mechanism, and the at least one of the operating system, firmware, and hardware is configured to provide for a periodic test to determine relative performance of different computer processing jobs on different ones of the computer processor cores.

7. A method for operating multiple processor cores, comprising:

obtaining a throughput metric that identifies throughput achieved by a plurality of computer processor cores as a function of workloads running on said computer processor cores, wherein the plurality of computer processor cores are on a single semiconductor die, in which at least two computer processor cores differ in processing capability, and wherein the computer processor cores execute the same instruction set; and

transferring individual ones of a plurality of computer processing jobs amongst targeted ones of said plurality of computer processor cores based on the throughput metric.

8. The method of claim 7, further comprising:

providing for a periodic test to determine relative performance of different computer processing jobs on different ones of the computer processor cores.

15. The method of claim 7, further comprising:

associating workloads for execution on specific processor cores based on annotations associated with the computer processing jobs.

18. The computer system of claim 1, wherein the performance measurement and transfer mechanism is configured to maximize total system throughput.

19. The computer system of claim 1, wherein the performance measurement and transfer mechanism is configured to transfer the executing computer processing jobs to a new assignment amongst the plurality of computer processor cores, collect performance statistics about execution at the new assignment, and then determine whether to reassign the executing computer processing jobs to different computer processor cores based on the performance statistics collected.

20. A computer system, comprising:

a plurality of computer processor cores in which at least two differ in processing performance, and wherein the plurality of computer processor cores are configured to execute the same instruction set; and

a performance measurement and transfer mechanism configured to move a plurality of executing computer processing jobs amongst the plurality of computer processor cores based on a measured throughput metric,

wherein the performance measurement and transfer mechanism is configured to swap execution of the executing computer processing jobs between the computer processor cores for a period of time, monitor resulting performance, and then build a data structure with relative performances of jobs on different types of the computer processor cores.

22. The computer system of claim 19, wherein the determination of whether to reassign the jobs to different computer processor cores also is based on at least one of a user-defined metric or a workload-defined metric.

23. The method of claim 7, wherein the throughput metric comprises a number of instructions per second.

24. The computer system of claim 1, wherein movement of the executing computer processing jobs is constrained to occur only at operating system time slice intervals.

25. A method for operating multiple processor cores, comprising:

assigning a plurality of computer processing jobs amongst a plurality of computer

processor cores, wherein at least two of the computer processor cores differ in size or complexity but execute the same instruction set, and

wherein assigning the plurality of computer processing jobs amongst the plurality of computer processor cores comprises matching requirements of the computer processing jobs to processing capabilities of the computer processor cores based on the sizes or complexities of the computer processor cores.

26. The method of claim 25, further comprising periodically testing to determine relative performance of different computer processing jobs on different ones of the computer processor cores.

29. A method for operating multiple processor cores, comprising:

obtaining a throughput metric that identifies throughput achieved by computer processor cores on a single semiconductor die as a function of workloads running on said computer processor cores; and

assigning a plurality of computer processing jobs amongst the computer processor cores based on the throughput metric, wherein at least two of the computer processor cores differ in size or complexity but execute the same instruction set;

transferring the computer processing jobs to a new assignment amongst the computer processor cores;

collecting statistics about execution performance of the computer processing jobs at the new assignment;

determining whether to reassign the computer processing jobs to different computer processor cores based on the statistics collected; and

building a data structure with relative performances of the computer processing jobs on different types of computer processor cores based on the statistics collected.

30. The method of claim 29, wherein the determination of whether to reassign the computer processing jobs to different computer processor cores also is based on at least one of a user-defined metric or a workload-defined metric.

31. (Previously Presented) The method of claim 29, wherein the throughput metric comprises a number of instructions performed per second.

32. The computer system of claim 1, wherein the processing capabilities of the computer processor cores are defined by one or more of chip area, available resource, and relative speed of the computer processor cores.

33. The computer system of claim 1, wherein the performance measurement and transfer mechanism is configured to move the plurality of executing computer processing jobs amongst the plurality of computer processor cores further based on annotations associated with the computer processing jobs.

34. The computer system of claim 1, wherein the performance measurement and transfer mechanism is configured to further re-assign the plurality of executing computer processing jobs amongst the plurality of computer processor cores by repeatedly performing a test to match the requirements of the plurality of executing computer processing jobs to the processing capabilities of the computer processor cores.

35. The method of claim 25, wherein assigning the plurality of computer processing jobs amongst the plurality of computer processor cores is further based on annotations associated with the computer processing jobs.

36. The method of claim 25, further comprising:

repeatedly performing a test to match requirements of the computer processing jobs to the processing capabilities of the computer processor cores; and

re-assigning the plurality of computer processing jobs amongst the plurality of computer processor cores based on the repeated tests.

37. The method of claim 29, wherein the throughput metric indicates total system throughput, and wherein the assigning maximizes the total system throughput, as indicated by the throughput metric.

IX. EVIDENCE APPENDIX

(1) Second Declaration Under 37 C.F.R. § 1.132 submitted with Amendment of May 10, 2010.

(2) Declaration Under 37 C.F.R. § 1.132 submitted with Amendment of October 2, 2009.

(3) U.S. Provisional Application Serial No. 60/409,105 (referenced in Amendment of May 10, 2010).

(4) U.S. Provisional Application Serial No. 60/409,106 (referenced in Amendment of May 10, 2010).

X. RELATED PROCEEDINGS APPENDIX

None.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Appl. No.:	10/621,067	§	Confirmation No.:	1252
Applicant:	Keith Farkas	§		
Filed:	07/16/2003	§		
TC/A.U.:	2195	§		
Examiner:	Kenneth Tang	§		
Title:	HETEROGENEOUS PROCESSOR CORE SYSTEMS FOR IMPROVED THROUGHPUT	§		
Docket No.:	200210109-1 (HPC.0762US)	§		

SECOND DECLARATION UNDER 37 C.F.R. § 1.132

We, Keith Farkas, Norman Paul Jouppi and Parthasarathy Ranganathan, state as follows:

1. We are the inventors of the subject matter of the present application (referenced above);

2. Rakesh Kumar and Dean M. Tullsen are not co-inventors of the present application.

3. At the time of the present invention, Keith Farkas was employed at Hewlett Packard. Norman Paul Jouppi is the Director of the Exascale Computing Lab at Hewlett Packard Labs. Parthasarathy Ranganathan is a Distinguished Technologist at Hewlett Packard Labs.

4. We conceived of the invention claimed in the present application and had a number of discussions regarding the subject heterogeneous multi-core processor technology before contacting Dean M. Tullsen and Rakesh Kumar.

5. While the invention had already been fully conceived and discussed, it was evident that simulations were appropriate to determine the degree of performance improvement to be achieved by the heterogeneous multi-core processor technology.

6. After we had conceived of the invention, we called Dean M. Tullsen and Rakesh Kumar on a confidential basis to discuss performing possible simulations of the invention.

7. Dean M. Tullsen is a professor at the University of California at San Diego, and at that time was an advisor to Rakesh Kumar, who was one of his doctoral students.

8. Rakesh Kumar, as primary researcher, performed the simulation of the heterogeneous multi-core processor technology revealed to Rakesh Kumar and Dean M. Tullsen by us.

9. After the simulations showed that a significant performance improvement could be achieved by the invention, Rakesh Kumar, Dean M. Tullsen, Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan began work on a paper titled "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures" for the IEEE Computer Architecture Letters ("Multi-Core Architecture Paper"), attached as Exhibit A. The Multi-Core Architecture Paper was initially submitted in March 2003 to the publishing entity (of the IEEE Computer Architecture Letters), and the final draft was submitted in April 2003.

10. As a result of the work performed during the simulations, various results were obtained, at least some of which were published in the Multi-Core Architecture Paper.

11. The content of the Multi-Core Architecture Paper relating to a heterogeneous multi-core processor that has multiple heterogeneous cores originated with or was obtained from Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan, inventors of the present application.

12. We also state that the authors of the Multi-Core Architecture Paper derived their knowledge of the subject matter described in the Multi-Core Architecture Paper relating to heterogeneous multi-core processors that have multiple heterogeneous cores from the inventors (including inventors Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan) of the present application.

13. We also state that the Multi-Core Architecture Paper describes the work of the inventors (including inventors Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan) of the present application.

14. We have reviewed Exhibit B, which purports to show a version of the Abstract of the Multi-Core Architecture Paper. The Abstract reflected in Exhibit B contains content from the Multi-Core Architecture Paper. Exhibit B erroneously shows that the Abstract of Exhibit B was published in January 2002, in volume 1, issue 1, of the IEEE Computer Architecture Letters. Exhibit B appears to be a summary provided by the following website: portal.acm.org. We believe that Exhibit B is in error, as it would have been impossible for the Abstract included in Exhibit B to have been published in January 2002. Based on our recollection, we did not first submit our paper (the Multi-Core Architecture Paper reflected in Exhibit A) to the publishing entity until March 2003. We also did not submit any Abstract relating to the Multi-Core Architecture Paper to the publishing entity in 2002. Therefore, any Abstract relating to the Multi-Core Architecture Paper would not have been published until March 2003 or later. Therefore, Exhibit B erroneously attributes a January 2002 publication date to the Abstract provided in Exhibit B.

15. Exhibit C shows the Table of Contents for the 2002 publication of the IEEE Computer Architecture Letters. Exhibit C does **not** contain any listing for Rakesh Kumar et al., "Processor Power Reduction Via Single-ISA Heterogeneous Multi-core Architectures."

16. Exhibit D shows the Table of Contents for the 2003 publication of IEEE Computer Architecture Letters. The Table of Contents in Exhibit D includes a listing for Rakesh Kumar et al., "Processor Power Reduction Via Single-ISA Heterogeneous Multi-core Architectures."

17. Exhibits C and D, which reflect information from the publisher of the Multi-Core Architecture Paper (Exhibit A) clearly indicate that the Multi-Core Architecture Paper was published in the 2003 publication of IEEE Computer Architecture Letters, not in the 2002 publication of the IEEE Computer Architecture Letters. Moreover, in view of the fact that we did not initially submit the Multi-Core Architecture Paper until March 2003, and in view of the fact that we did not submit any Abstract

relating to the Multi-Core Architecture Paper in 2002, it would have impossible for any version of the Abstract relating to the Multi-Core Architecture Paper to be published in 2002. Therefore, it is our belief that Exhibit B contains erroneous information, since any Abstract of the Multi-Core Architecture Paper would not have been published until 2003.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

4/27/2010
DATE

KEITH FARKAS

DATE

NORMAN PAUL JOUPPI

DATE

PARTHASARATHY RANGANATHAN

relating to the Multi-Core Architecture Paper in 2002, it would have impossible for any version of the Abstract relating to the Multi-Core Architecture Paper to be published in 2002. Therefore, it is our belief that Exhibit B contains erroneous information, since any Abstract of the Multi-Core Architecture Paper would not have been published until 2003.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

DATE

KEITH FARKAS

4/20/2010
DATE

Norman P. Jouppi
NORMAN PAUL JOUPPI

4/20/2010
DATE

PARTHASARATHY RANGANATHAN

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Appl. No.:	10/621,067	§	Confirmation No.:	1252
Applicant:	Keith Farkas	§ §		
Filed:	07/16/2003	§		
TC/A.U.:	2195	§ §		
Examiner:	Kenneth Tang	§		
Title:	HETEROGENEOUS PROCESSOR CORE SYSTEMS FOR IMPROVED THROUGHPUT	§ § § § §		
Docket No.:	200210109-1 (HPC.0762US)	§ §		

DECLARATION UNDER 37 C.F.R. § 1.132

We, Keith Farkas, Norman Paul Jouppi and Parthasarathy Ranganathan, state as follows:

1. We are the inventors of the subject matter of the present application (referenced above);

2. Rakesh Kumar and Dean M. Tullsen are not co-inventors of the present application.

3. At the time of the present invention, Keith Farkas was employed at Hewlett Packard. Norman Paul Jouppi is the Director of the Exascale Computing Lab at Hewlett Packard Labs. Parthasarathy Ranganathan is a Distinguished Technologist at Hewlett Packard Labs.

4. We conceived of the invention claimed in the present application and had a number of discussions regarding the subject heterogeneous multi-core processor technology before contacting Dean M. Tullsen and Rakesh Kumar.

5. While the invention had already been fully conceived and discussed, it was evident that simulations were appropriate to determine the degree of performance improvement to be achieved by the heterogeneous multi-core processor technology.

6. After we had conceived of the invention, we called Dean M. Tullsen and Rakesh Kumar on a confidential basis to discuss performing possible simulations of the invention.

7. Dean M. Tullsen is a professor at the University of California at San Diego, and at that time was an advisor to Rakesh Kumar, who was one of his doctoral students.

8. Rakesh Kumar, as primary researcher, performed the simulation of the heterogeneous multi-core processor technology revealed to Rakesh Kumar and Dean M. Tullsen by us.

9. After the simulations showed that a significant performance improvement could be achieved by the invention, Rakesh Kumar, Dean M. Tullsen, Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan began work on a paper titled "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures" for the IEEE Computer Architecture Letters ("Multi-Core Architecture Paper"), attached as Exhibit A. The Multi-Core Architecture Paper was submitted in March 2003, and the final draft was submitted in April 2003.

10. As a result of the work performed during the simulations, various results were obtained, at least some of which were published in the Multi-Core Architecture Paper.

11. The content of the Multi-Core Architecture Paper relating to a heterogeneous multi-core processor that has multiple heterogeneous cores originated with or was obtained from Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan, inventors of the present application.

12. We also state that the authors of the Multi-Core Architecture Paper derived their knowledge of the subject matter described in the Multi-Core Architecture Paper relating to heterogeneous multi-core processors that have multiple heterogeneous cores from the inventors (including inventors Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan) of the present application.

13. We also state that the Multi-Core Architecture Paper describes the work of the inventors (including inventors Keith Farkas, Norman Paul Jouppi, and Parthasarathy Ranganathan) of the present application.

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true: and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

9/29/2009
DATE

KEITH FARKAS

DATE

NORMAN PAUL JOUPPI

DATE

PARTHASARATHY RANGANATHAN

U.S. Serial No. 10/621,067
Declaration Under 37 C.F.R. § 1.132

I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code, and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

DATE

KEITH FARKAS

9/29/2009
DATE

Norm P. Jouppi
NORMAN PAUL JOUPPI

9/29/2009
DATE

PARTHASARATHY RANGANATHAN

METHOD AND PROCESS FOR DYNAMIC TRACKING AND PREDICTING OF PHASE-BASED BEHAVIOR

The goal of this invention is to provide a unified profiling on-the fly algorithm for hardware or software that can efficiently capture, classify, and predict program behavior all at run-time with little or no support from software. Our invention is a technique for tracking a metric which is a very good indicator of the proportion of instructions that were executed from different sections of code, so that we can find generic phases that correspond to changes in behavior across many metrics. By classifying phases generically, we avoid the need to identify phases for each optimization, and enable a unified prediction scheme that can forecast future behavior. We examine the ability of our phase tracking architecture to accurately capture the phase behavior of a program's execution with respect to its overall performance (IPC), branch prediction, cache performance, and energy, and show how phase behavior may be captured efficiently using a simple predictor.

The invention focuses on a run-time algorithm for finding phase-based behavior and predicting phase changes. This algorithm can be used either in software or hardware, and the attached paper provides a hardware implementation. It is unique in that there has been no prior on-the fly algorithm for performing phase tracking or prediction of program behavior.

The unique claims/ideas in this work area:

(1) A novel metric which is easy to gather in hardware but still captures enough information to allow for phase-based behavior analysis.

(2) An efficient implementation of hardware that captures said metric

(3) A hardware architecture for predicting program phases

Dhodapkar and Smith independently (at the same time) proposed a run-time/hardware approach for identifying when a phase change occurs in the *working set* behavior. Their goal was to capture the working set size and behavior of programs dynamically and efficiently for a given architectural optimization. They examined capturing this information for the instruction cache, and dynamically reconfiguring the cache based on its working set size to save power. Their scheme focuses on developing an algorithm for detecting a working set change, specific to instruction cache behavior.

Managing Multi-Configuration Hardware via Dynamic Working Set Analysis Ashutosh S. Dhodapkar and James E. Smith 29th International Symposium on Computer Architecture (ISCA 2002), Alaska; May 2002.

Our research makes different contributions then the work by Dhodapkar and Smith. A goal of our approach is to develop a generic fast run-time algorithm for classifying phases, and identifying phase changes that relies only on dynamic code profile information. Our phase-based tracker can then be used for tracking phase behavior for many different types of hardware metrics. In addition, our technique can be used to predict phase changes and the expected metrics/performance for those phase changes.

Hardware phase classifier for:

A) Voltage scaling

B) Gating hardware structure usage

C) Orchestrating Multithreaded Execution

D) Guiding software policy in threaded architectures

E) Monitoring program behavior for further dynamic optimization

1 Introduction

Modern processors can execute upwards of 5 billion instructions in a single second, yet most architectural features are targeted at the behavior of programs on a timescale of hundreds to thousands of instructions, less than half a μS. While these optimizations can provide large amounts of benefit, they are limited in their ability to see the program behavior in a larger context.

Recently there has been a renewed interest in examining the run-time behavior of executing programs over longer periods of time [6, 10, 15, 16]. It has been shown that programs can have considerably different behavior depending on which portion of execution is examined. More specifically it has been shown that many programs execute as a series of phases, where each phase may be very different from the others, yet still has a fairly homogeneous behavior within the phase. Taking advantage of this time varying behavior can lead to, among other things, improved power management, cache control, and more efficient simulation. The primary goal of this research is the development of a unified run-time phase detection and prediction mechanism that can be used to guide any optimization seeking to exploit large scale program behavior.

A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being an interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency. Simply put, if a phase of execution is correctly identified there should be only small amounts of variation between any two segments of that phase. A key point of this paper is that the phase behavior seen in any program metric is directly a function of the application's behavior (the code being executed), and if we can accurately capture this behavior at run-time through the computation of a *single* metric, we can use this to guide many optimization and policy decisions without duplicating phase detection mechanisms for *each* optimization.

In this paper we present an efficient run-time phase tracking architecture that is based on detecting changes in portions of code executed. Since the implementation is based upon the code execution frequencies, it is independent of any individual metric. Even so, we show that the changes in many important metrics, such as IPC and energy, correlate very closely with changes in our metric. This allows our phase tracker to be used as a general technique for correlating program behavior with any code executing. We evaluate the effectiveness of the hardware based phase detection and identification mechanisms and show how they can be used to automatically partition the behavior of the program into homogeneous sections of execution.

In addition, we present a novel phase prediction architecture that can predict not only when a phase change is about to occur, but to which phase it is about to transition. The predictor is based on the run-length encoding of past phase information to create a history based on both phase pattern and duration. Because the predictions are generic phase identifiers, this information can then be used to make predictions about future of other metrics such as power and performance.

To summarize, the contributions of this paper are:

- An efficient unified phase tracking architecture that is based upon examining program execution frequencies at run-time. Since the implementation is based upon the code executed, it is independent of the different hardware metrics such as IPC and power, but can be used to identify changes in execution.
- A novel phase change prediction architecture that compresses the past history of phases and uses this to predict the next phase to occur.
- We show that a program's phases are accurately classified using our generic phase tracker architecture across several metrics including IPC, energy, I-cache and D-cache miss rates, and branch misprediction rates.

The rest of the paper is laid out as follows. Section 2 explains the motivation behind our phase tracking and prediction architecture. In section 3, prior work related to phase-based program behavior are discussed. Simulation methodology and benchmark descriptions can be found in section 4 while section 5 details it's implementation. The design and evaluation of the phase predictor can be found in section 6. The results are summarized in section 7.

2 Motivation

With the advent of simultaneous multithreading (SMT), managed run-times, virtualized hardware, and dynamic voltage scaling, the traditional strict chain of command over the execution environment is changing. No longer is the operating system able to dictate the efficient behavior of the processor without feedback from the underlying hardware. Instead, a cooperative environment is needed; the operating system informing the processor of the needs of the user, and the processor informing the operating system of the behavior of the executing threads [9, 12, 21].

For example, in SMT enabled architectures [19], the hardware presents the opportunity to schedule two or more threads to execute concurrently sharing the underlying hardware. In their work on symbiotic scheduling, Snavely and Tullsen [18] show that if adequate knowledge of program behavior is gathered, threads can be scheduled such that the use of resources can be utilized much more effectively.

In systems with dynamic voltage scaling [4], the clock rate of the processor is slowed down so that the voltage may be throttled back. While this results in a longer execution time, the total amount of energy used by the processor can be significantly reduced because of the non-linear relationship between voltage and cycle time. Optimizations of this sort also require both operating system and processor support [8]. Not only must the processor be able to scale back the voltage and clock rate dynamically, it must be able to monitor the behavior of the code so that it can more efficiently schedule voltage shifts. There have been several papers which address the idea of adaptive power control [2, 6, 8, 10, 20] in various forms, and each relies on reacting to observed, profiled or estimated usage of various structures.

One possible problem with close interaction between processor and operating system is that the interface becomes more complicated, which requires greater amounts of cooperation between processor designers and operating system writers. One solution to this problem is the creation of managed run-times and virtualized hardware which seek to define an interface for the operating system software to interact above the level of a traditional instruction set architecture [1, 6, 7, 9, 11, 13, 14].

The goal of this research is to design an efficient and general purpose technique for capturing and predicting the run-time phase behavior of programs to help guide any optimization seeking to exploit large scale program behavior. Figure 1 helps to motivate our approach to the problem. This figure shows the behavior of two programs, `gcc` and `gzip`, as measured by various different statistics over their execution. The runs for `gcc` and `gzip` are shown from start to finish. Each point on the graph is taken over 10 million instructions worth of execution. The metrics shown are, the number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction (il1) missed, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC. The results show that all of the metrics change in unison, but not necessarily in the

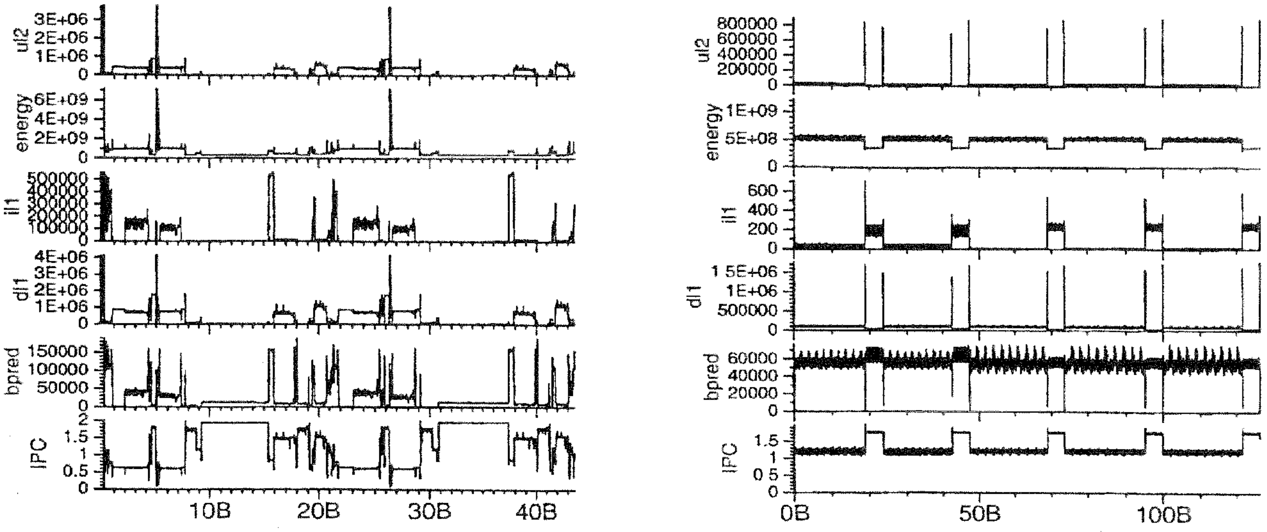


Figure 1: To illustrate the point that phase changes happen across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the programs gcc (shown left) and gzip (shown right). Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction (il1) missed, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.

same direction. In addition to this, patterns can be seen in the execution of these programs on a very large time scale.

3 Related Work

In [15], Sherwood and Calder provided results showing how the phase behavior of programs vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy for all the SPEC 95 programs. When looking at these metrics, they found that many programs have repeating patterns, and that the patterns affected all of the metrics at the same time. In this way the execution was divided up into phases.

In [16] and [17], Sherwood et. al. presented the idea of using the code executed in the program to identify different phases in a program's execution to improve simulation efficiency and accuracy. Their goal was to identify these phases, and then to pick a simulation point from each phase. Their simulation points would then be used for detailed pipeline simulation providing a representative execution of the whole program by simulating a single point from each representative phase of execution. They choose to focus on identifying the different phases based on the code executed, and not a hardware metric in order to quickly determine where to simulate. Our approach is motivated by this work, and our architecture identifies the phase change behavior in hardware based upon changes in executed code. The prior work by Sherwood et. al. did not examine a run-time/hardware algorithm for identifying phases and phase changes, which is the focus of our research along with creating an efficient and accurate phase predictor.

In [17], Sherwood et. al. make use of *Random Projections* of the data to reduce the dimensionality of the samples being taken. A random projection takes trace data in the form a matrix of size $L \times B$, where L is the length of the trace and B is the number of unique basic blocks, and multiplies it by a random matrix of size $B \times N$, where N is desired dimensionality of the data which is much smaller than B. This creates a new matrix of size $L \times N$ which has clustering properties very similar to the original data. The random projection technique is a state of the art technique in clustering algorithms, and its applicability to capturing phase behavior was shown in [17]. We make use of this idea of projecting the executed code into our phase identifier via hashing as described in the Section 5.

Dhodapkar and Smith [6] just independently proposed a run-time/hardware approach for identifying when a phase change occurs in the working set behavior. Their goal was to capture the working set size and behavior of programs dynamically and efficiently for a given architectural optimization. They examined capturing this information for the instruction cache, and dynamically reconfiguring the cache based on its working set size to save power. Their scheme focuses on developing an algorithm for detecting a working set change, which can then be applied to a variety of memory structures. A goal of our research is to develop a generic fast run-time algorithm for identifying phase changes that relies only on dynamic code profile information. Our phase-based tracker can then be used for tracking phase behavior for many different types of hardware metrics, and can be use in conjunction with the Dhodapkar and Smith approach.

Another goal of our architecture is to be able to predict what the next phase will be, and we provide a novel architecture that can fairly accurately predict what the next phase will be, along with predicting when there will be a phase change. In comparison, Dhodapkar and Smith do not present a phase based predictor but concentrate on the detection of phases changes for reactive optimization.

Instruction Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
Unified L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 latency
Branch Predictor	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 4 operations per cycle, 64 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

4 Methodology

To perform our study, we collected information for five of SPEC 2000 programs `bzip-graphic`, `gcc-166`, `gzip-graphic`, `mesa-ref`, and `vpr-place`, all with reference inputs. All programs were executed from start all the way through *to completion* using SimpleScalar and Wattch. Because of this lengthy simulation time incurred by executing all of the programs to completion, we chose to focus on only 5 programs. The programs `bzip`, `gcc`, and `gzip` were choose because they were found to have the most time varying behavior, and thus make them difficult examples to attempt to capture. The programs `mesa` and `vpr` are included as examples of programs that have more regular, and non-time varying behavior respectively.

Each program was compiled on a DEC Alpha AX P-21164 processor using the DEC C, C++, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used is derived from the SimpleScalar 3.0b tool set [5], a suite of functional and timing simulation tools for the Alpha AXP ISA. The baseline microarchitecture model is detailed in Table 1. In addition to this, we wanted to examine energy usage results, so we used a version of Wattch [3] to capture that information. We modified all of these tools to log and reset the statistics every 10 million instructions, and we use this as a base for evaluation.

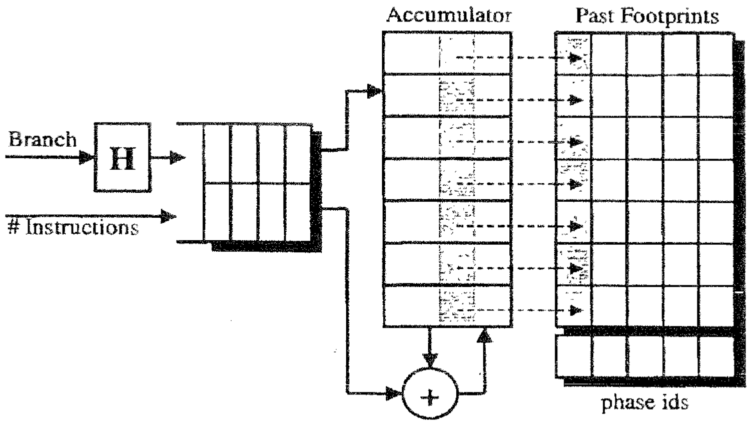


Figure 2: Our phase profile generation architecture. Each branch PC is captured along with the number of instructions from the last branch. The bucket entry corresponding to a hash of the branch PC is incremented by the number of instructions. After the profiling interval has completed, this information is then classified, and if it is found to be unique enough, stored in the past footprint table along with it's classification id.

5 Phase Capture

In this section we describe our architecture for capturing phase based behavior and examine the accuracy of using the program behavior for our architecture to identify phase changes for other hardware metrics.

5.1 Profiling

Our phase tracker architecture needs to work at two different time-scales, it needs to be able to capture profiles very quickly in order to keep up with processor speeds, while the other time-scale is that of performing phase analysis which needs to be done only every profile interval, which we assume to be 10 million instructions for this paper.

Our phase profile generation architecture can be seen in Figure 2. The idea is to capture basic block information during execution, while not relying on any compiler support. Basic blocks which are larger need to be weighed more heavily as they account for a more significant portion of the execution. To approximate the capturing of basic block information, we capture branch PCs and the number of instructions executed between branches. The input to the architecture is a tuple of information, a branch id (PC) and the number of instructions since the last branch PC was executed. This allows us to roughly capture each basic block executed, along with the weight of the basic block in terms of the number of instructions executed as was done by Sherwood et. al. [16, 17] for identifying simulation points.

The branch id is then reduced to a number from 1 to *Nbuckets* using a hash function. For our hash function, we have found that 20 buckets was enough to distinguish between all of the different phases for even some of the more complex programs such as gcc. A counter is kept for each bucket, and the counter is incremented by the number of instructions from the last branch to the current branch being processed. Each accumulator table entry is a large (in this study 24-bit) saturating counter, which will most likely not saturate during our profiling interval of 10 million instructions. Updating the accumulator table is the only operation that needs to be performed at a rate equivalent to the processor's execution of the program. In comparison, the phase classification described below, needs to only be performed every 10 million instructions (at the end of each interval), and thus is not nearly as performance critical.

The hash function we use for the PC is basically a random projection as described in Section 3. If a random projection matrix is chosen such that all of the elements of the matrix are either 0 or 1, and it is chosen such that no column of the matrix contains more than a single 1, then the random projection can be implemented as simply a hashing mechanism. In other words one could think of this as generating a random bit mask that filters all but one of the buckets at each step of the hashing operation. We have designed our phase capture mechanism around this principal.

Figure 3 shows the effect of tracking the behavior of programs using phase tracker for gzip. The x-axis of the figure is in billions of instructions, just as is the case in Figure 1. Each point on the y-axis represents an entry of the phase tracker's accumulator table. Each point on the graph corresponds to the value of the corresponding accumulator table entry at the end of a profiling interval. Dark values represent high execution frequency, while light values correspond to less. The same trends that were seen in Figure 1 for gzip can be clearly seen in Figure 3. In both of these figures, we can see at the highest level of granularity that there are at least three different phases labeled A, B and C. In Figure 3, the phase tracker table entries 2, 13 and 17 and to some degree 5 distinguish the two identical long running phases labeled A from a group of three long running phases labeled C. Phase table entries 12 and 20 clearly distinguish phase B from both A and C. This pictorially shows that the phase tracker should be able to break the the programs execution into the corresponding phases based upon just the executed code, and these phases correspond to the behavior seen across the different program metrics in Figure 1.

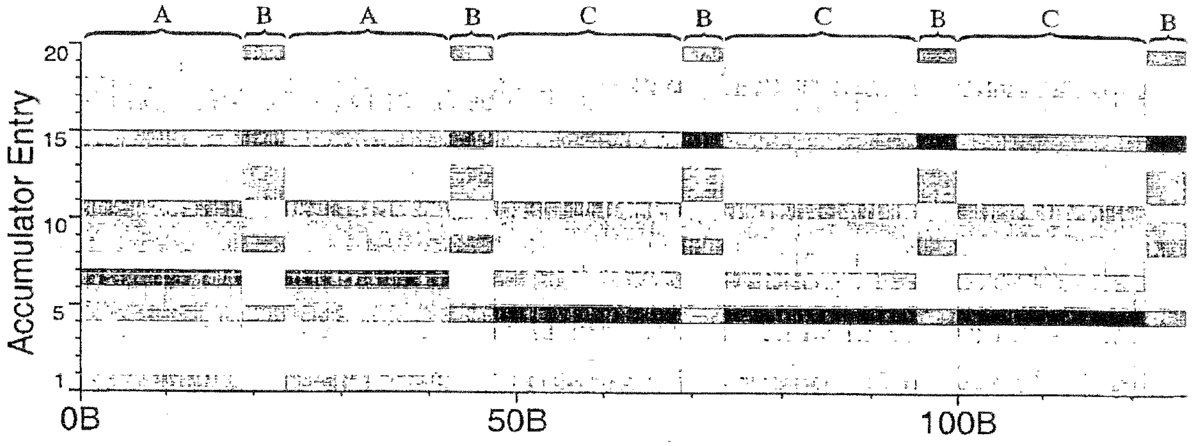


Figure 3: Visualization of the accumulator table used to track program behavior on the program `gzip`. The x-axis is in billions of instructions, while the y-axis is the entry of the accumulator table. Each point on the graph corresponds to value of the accumulator table at the end of a profiling interval where dark values correspond to more heavily accessed entries. The same trends that were seen in Figure 1 can be clearly seen in Figure 3.

5.2 Classification

After the profiling interval has elapsed, the phase must then be classified. In order to do this we must keep around a history of past phase information.

If we fix the number of instructions for a profiling interval then we know that by dividing each bucket by this fixed number we get a percentage of execution that was accounted for by all instructions mapped to that bucket. However, we do not need to know the exact percentages for each bucket. Instead of keeping the full counter values, we can instead compress phase information down to a couple of the most significant bits. This compressed information will then be kept in the Past Footprint table as shown in Figure 2.

The number of bits of information that we need to keep is related to $Nbuckets$. As we increase the number of buckets, the data is spread over more buckets (table entries), and thus we will need higher precision to identify the greater differences seen between the values in different buckets. In order to be safe, we would like *any* distribution into buckets to provide useful information. In order to do this we need to insure that even if data is distributed perfectly evenly over all of the buckets, that we still record information about the frequency of those buckets. This can be achieved by reducing the accumulator counter by:

$$(bucket[i] \times Nbuckets)/(intervalsize)$$

If the number of buckets and interval size are powers of two, this means a simple shift operation. For the number of buckets we have chosen (32), and the interval size we profile over, this reduces the bucket size down to 6 bits, an requires 24 bytes of storage for each unique phase in the Past Footprint table.

After we have reduced the vector, we now begin the classification process by comparing it to a set of representative past footprint vectors. For each vector in the table starting with the oldest, we compare the current vector to that vector. We describe in the next section how we perform the comparison and determine what a match is. If their is a match, we classify the profiled section of execution into the same phase as the past footprint vector, and the current vector is not inserted into the past footprint table. If there is no match, then we have just seen a new phase and hence must create a new unique phase ID into which we can classify it. This is done by choosing a unique phase ID out of a fixed pool of a maximum number of IDs. We then allocate a new past footprint entry, and store with that entry the newly allocated phase ID and we also store the current vector into that footprint entry. This allows future similar phases to be classified with the same ID. In this way only a single vector is kept for each unique phase id, to serve as a representative of that phase. After a phase ID is provided, for the most recent interval, it is passed along to prediction and statistic logging, and the phase identification part of our algorithm is complete.

Figure 4 shows the number of phase IDs found for each of the programs and the percent of execution they accounted for. The results show that most of the program's behavior can be captured using relatively few phases. The y-axis is the percent of program execution covered, and the x-axis is the minimum number of phases needed to capture that much program execution. The phases were identified and then sorted by size. For example, the program `vpr` is fairly homogeneous and by capturing a single phase the entire program behavior can be captured.

For `gzip`, in Figures 1 and 3 we saw that there are 3 high level phases of execution, but in Figure 4 we show that 6 phase IDs are needed to capture 90% of the programs execution. This is because there are actually some smaller recurring behavior going on within the large phases `A` and `C`, and our phase tracker classification breaks those into a few smaller phases. Even so, we note that our phase tracker classified all of the execution of `A` into separate phase IDs from `C`.

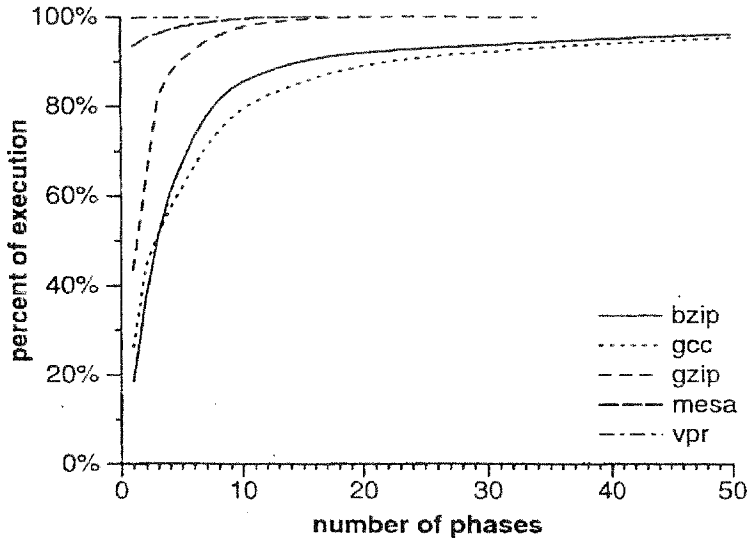


Figure 4: Graphs of the minimum number of phases that need to be captured versus the amount program execution they cover. The y-axis is the percent of program execution that is covered. The x-axis is the minimum number of phases of phases need to capture that much program execution.

5.3 Finding a Match

In the above section, we explain that we search through the history to find a match, but what we search for is actually complicated by the fact that the match we are searching for is not necessarily an exact match. Two sections of execution that have very close footprints could easily be considered a match, even if they do not compare exactly. To compare two vectors to one another, the above algorithm uses the Manhattan-distance between the two, which is the element-wise sum of differences.

If we set the threshold too low, the phase detection will be overly sensitive, and will classify the program into many very tiny phases which will cause us to lose any benefit from doing run-time phase analysis in the first place. If you set the threshold too high, the classifier will make not be able to distinguish between phases with different behavior. To quantify this effect, the threshold choice has on the overall effectiveness of the approach we examined a range of thresholds from 1, which is requires a perfect match, to 20, which is almost half of the maximum allowable difference.

To evaluate the effectiveness of our phase tracker, we examine how well it tracks the phase changes in IPC. Data

Cache miss rate, Energy, and Level 2 Cache miss rate. We define an actual phase shift as a change in any measured program metric of more than 10%. If any of the above metrics change by more that 10% then this is considered as a phase change.

We first examine the percentage of actual phase changes that occurred in the metrics that our phase tracker also correctly identified as a phase change (using only the program distribution frequencies). We call this number the capture rate. Figure 5 shows this statistic while varying the distance threshold. We see that as we increase the threshold, and hence decrease the sensitivity, we capture less and less of the actual phase shifts in the program. We see that using a threshold of 6 provides us with a phase capture rate above 90% for all programs.

We now examine the percent of false positives identified by our phase tracker. These are phase changes that are predicted by our phase tracker that did not occur in the three metrics we are examining. This does not necessarily mean that there was no change in behavior, but for the purposes of this study we will assume that this is the case in order to be conservative. Figure 6 shows the percent of false positives that occur varying the threshold. It shows that as we increase the required threshold, which in turn reduces the sensitivity, the false positives significantly decrease. With a threshold of 10, we see that all of the programs have less than 10% false positives, and for a threshold of 6 all but one program `gzip` has less than 10% false positives.

In order to strike a balance between having a high capture rate and reducing the percent of false positives we chose to use a threshold of 6 for the rest of the results in this paper. We feel that it is better to divide up the number of phases over zealously, than it is to miss an actual phase transition. With a threshold of 6, our phase tracker misses less than 5% of the metric-based phase transitions for all of the programs except `gcc`, where we miss less than 10%.

5.4 Results

By using the techniques presented above, we can now perform phase classification on programs at run-time with little to no impact on the design of the processor core. The goal of phase classification, as stated in Section 2, is to divide up the program into a set of phases that are fairly homogeneous. This means that an optimization adapted to and applied to a single segment of execution that is taken from that phase, will apply equally well to the other parts of the phase. In order to quantify the extent to which we have achieved this goal, we need to examine the statistics on a *per-phase* basis, as well as examine the homogeneity of the statistics from each phase.

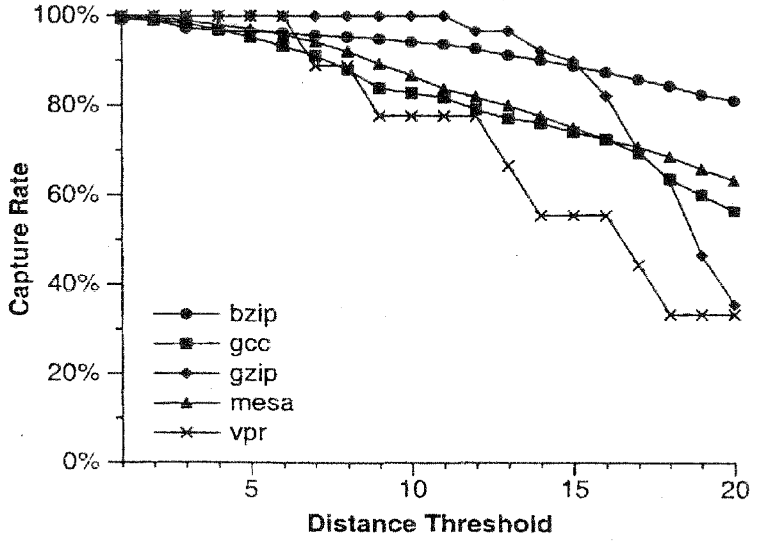


Figure 5: The effect of adjusting the distance threshold on the capture rate of phase change identification. For a threshold of 6, we capture over 90% of phase changes for all programs.

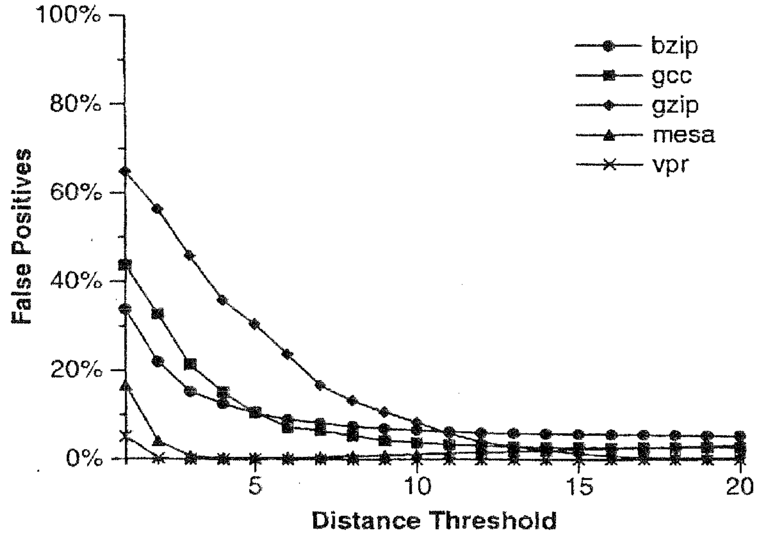


Figure 6: The effect of adjusting the distance threshold on the percent of false positives found.

	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
bzip	full	1.52	(27.8%)	41290	(101.6%)	55526	(185.5%)	3	(398.3%)	4.70E+08	(47.6%)	34659	(200.4%)
	15.6%	1.44	(10.7%)	30391	(37.8%)	29181	(37.4%)	0.14	(1644.1%)	4.41E+08	(11.7%)	16786	(40.9%)
	13.8%	1.94	(0.0%)	1660	(52.1%)	2493	(0.9%)	0.37	(699.6%)	3.23E+08	(0.1%)	1247	(1.0%)
	9.5%	1.67	(0.3%)	19636	(15.0%)	15267	(2.9%)	2.91	(330.2%)	3.75E+08	(0.3%)	7222	(2.3%)
	8.9%	1.85	(1.6%)	72235	(17.4%)	1092	(21.0%)	0.00	(3062.7%)	3.39E+08	(1.6%)	498	(19.9%)
	6.1%	1.82	(1.8%)	82469	(17.3%)	1291	(25.7%)	0.01	(1910.8%)	3.44E+08	(1.8%)	585	(25.4%)
gcc	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
	full	1.32	(43.4%)	27741	(135.5%)	445083	(110.7%)	50763	(203.2%)	6.44E+08	(90.0%)	227912	(139.7%)
	18.5%	0.61	(1.6%)	34665	(22.0%)	753382	(5.4%)	125091	(23.2%)	1.03E+09	(1.8%)	395997	(5.3%)
	18.1%	1.95	(0.3%)	13048	(3.9%)	28112	(15.1%)	43	(73.9%)	3.22E+08	(0.2%)	1006	(5.6%)
	7.2%	0.64	(0.2%)	843	(15.1%)	885081	(0.1%)	75	(215.5%)	9.78E+08	(0.3%)	443655	(0.1%)
	4.0%	1.49	(1.2%)	10145	(7.6%)	703554	(6.8%)	15591	(5.2%)	4.20E+08	(1.1%)	354084	(7.0%)
	3.9%	1.76	(1.6%)	2015	(13.6%)	98947	(5.9%)	102	(45.1%)	3.57E+08	(1.6%)	15595	(12.6%)
gzip	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
	full	1.33	(16.3%)	56045	(11.1%)	90446	(58.2%)	60	(138.1%)	4.82E+08	(13.5%)	22880	(112.0%)
	17.1%	1.24	(3.4%)	53300	(10.8%)	96960	(10.1%)	12	(44.2%)	5.05E+08	(3.5%)	24218	(8.6%)
	9.4%	1.23	(3.8%)	54973	(11.5%)	99523	(11.3%)	11	(45.5%)	5.09E+08	(3.8%)	24518	(9.3%)
	8.8%	1.76	(0.6%)	56449	(4.8%)	37331	(5.6%)	241	(8.4%)	3.55E+08	(0.6%)	5617	(15.6%)
	8.0%	1.22	(4.3%)	54791	(6.8%)	99671	(11.9%)	40	(25.7%)	5.14E+08	(4.4%)	28153	(11.0%)
	7.4%	1.24	(3.1%)	55215	(11.1%)	96701	(9.6%)	12	(35.4%)	5.04E+08	(3.2%)	23701	(8.4%)
mesa	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
	full	1.94	(12.4%)	39280	(10.6%)	27155	(143.6%)	10585	(26.3%)	3.35E+08	(29.0%)	8625	(233.4%)
	68.0%	1.99	(1.7%)	40200	(3.6%)	18929	(10.8%)	10695	(26.2%)	3.15E+08	(1.7%)	4571	(39.8%)
	18.6%	2.00	(1.7%)	39951	(3.6%)	18659	(11.9%)	10192	(22.2%)	3.13E+08	(1.7%)	3919	(45.8%)
	5.5%	2.01	(1.3%)	40067	(3.5%)	18818	(13.8%)	11918	(20.1%)	3.11E+08	(1.3%)	3265	(47.7%)
	1.7%	0.71	(1.5%)	17078	(6.7%)	250052	(4.2%)	6281	(24.7%)	8.83E+08	(1.4%)	123618	(4.7%)
	1.1%	2.01	(1.5%)	39686	(3.5%)	21664	(40.3%)	10926	(18.4%)	3.13E+08	(1.6%)	4373	(107.0%)
vpr	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
	full	0.71	(5.4%)	111808	(3.1%)	186675	(4.2%)	28	(1022.8%)	8.88E+08	(3.0%)	111813	(6.9%)
	99.8%	0.71	(2.1%)	111871	(2.8%)	186942	(2.3%)	17	(259.7%)	8.89E+08	(2.0%)	111989	(5.8%)
	0.1%	1.58	(4.9%)	85489	(3.8%)	24127	(32.5%)	7578	(11.9%)	3.98E+08	(4.9%)	7477	(51.6%)
	0.0%	1.76	(0.6%)	44251	(4.4%)	7139	(0.5%)	0	(0.0%)	3.55E+08	(0.0%)	2413	(0.0%)
	0.0%	0.72	(0.0%)	60167	(0.0%)	207730	(0.0%)	792	(0.0%)	8.67E+08	(0.0%)	112296	(0.0%)

Figure 7: Examination of per-phase homogeneity compared to the program as a whole (denoted by `full`). For each full program and each of the top 5 phases of each program, we show the average value of each metric and the standard deviation. The name of the phase is the percent of execution that it accounts for in terms of instructions. These results show that after dividing up the program into phases using our run-time scheme the behavior within each phase is quite consistent across most benchmarks.

Figure 7 shows the results of performing this analysis on the phases as determined at run-time. It shows for each program a set of statistics and their values for each phase. The first phase that is listed (separated from the rest) is full, which is the result if you classify the entire program into one single phase. The results show that for gcc for example, the average IPC of the entire program was 1.32, while the average number of caches misses was about 445,000 per ten million instructions. In addition to just the average value we also show the standard deviation for that statistic. For example, while the average IPC was 1.32 for gcc, it varied with a standard deviation of over 43% from interval to interval. If we do a good job of classifying the phases, the standard deviations should be low for a given phase ID.

Underneath the phase marked full, is the top 5 largest phases from the program as identified by our phase tracker. The phases are weighted by how much of the program's executed instructions they account for. For gcc the largest phase accounts for 18.5% of the instructions in the entire program and has an average IPC of 0.61 and a standard deviation of only 1.6% (of 0.61). The other top 4 phases have standard deviations at or below this level, which means that our technique was successful at dividing up the execution of gcc into large phases with similar execution behavior with respect to IPC.

The statistics examined are IPC, number of branch mispredictions (bpred), number of data cache misses (dl1), number of instruction cache misses (dl1), energy, and number of misses to second level data cache (ul1). One thing that stands out is the high levels of deviation (in terms of percent) that can occur for some metrics. For instance, if you examine the standard deviation in instruction cache miss rate for bzip you might first be shocked by the 1644% standard deviation, but if you consider that the average instruction miss rate for that phase is less than 0.14 misses every ten million instructions, then it makes sense that this number might fluctuate between 0 and 2 misses per ten million instructions. Despite this exaggeration of small error in very rarely occurring events, we believe that percent standard deviation is an excellent way of examining how good of a job we are doing.

If you look at energy consumption of gcc, you can see that energy consumption swings radically (a standard deviation of 90%). This can be seen visually back in Figure 1, which plots the energy usage versus instructions executed. However after dividing up into phases, we see that each phase has a very little variation within itself, all have standard deviations less that 2%. It can also be seen looking at gcc that the phase partitioning does a very good job across all of the measured statistics even though only *one* classification in made and used for all metrics. This indicates that the phases that we have chosen are in some way representative of the actual behavior of the program.

6 Phase Prediction

The prior section described our phase tracking architecture, and how it can be used to classify phases. In this section we focus on using this phase information to predict the next phase to occur. It is important for a variety of applications to be able to predict future phase changes so that the system can configure itself for the code it is now executing rather than simply reacting to a change in behavior.

Phase behavior is quite stable for some programs. Programs such as `vpr` (see figure 4) are examples of phase stable programs, and predicting future behavior is not all that difficult because it is most likely exactly the same as what was just executed. Other programs are not so easy to deal with. A perfect example of this is `bzip` which has repeating phase behavior larger than 10 million instructions. In this case, you get a large repeating pattern, where no two phases executing sequentially are that similar, but there is a larger order to the sequence. By adding in a prediction scheme for these cases we can not only take advantage of stable conditions as in past research, but actually take advantage of any repeating patterns in program behavior.

The prediction of phase behavior in programs requires a new predictor. After observing the way that phases change in our chosen set of applications, we determined that two things were important. First, the set of phases leading up to the prediction is very important, and second, the duration of execution of those phases is important. A classic prediction model that is easily implementable in hardware is a Markov Model. Markov Models have been used in computer architecture to predict prefetch addresses and branches in the past. The basic idea behind a Markov Model is that the next state of the system is related to the last N past states.

Figure 8 shows our prediction architecture. This is a *Run Length Encoding* (RLE) Markov predictor. The basic idea behind the predictor is that it uses a run-length encoded version of the past history to index into a prediction table. The index into the prediction table is a hash of the phase identifier and the number of times that phase identifier has occurred in row. Figure 8 shows how this can be computed.

The prediction table is further optimized by adding some hysteresis to the table to prevent noise from affecting the result too strongly. Instead of storing just the last phase identifier that was found to follow the history, we

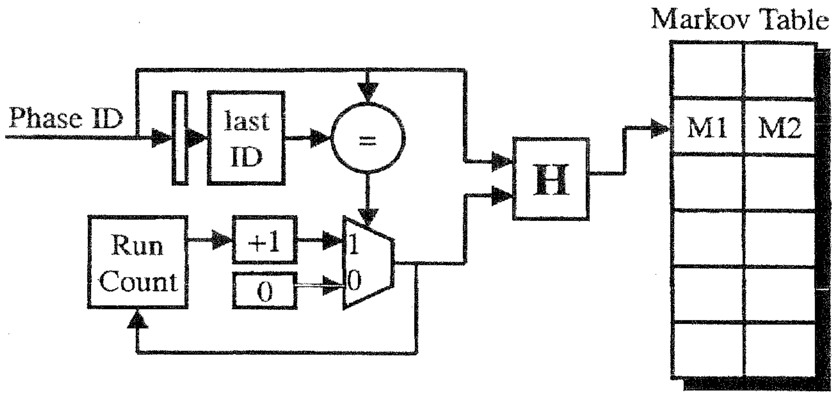


Figure 8: Phase Prediction Architecture for the Run Length Encoded (RLE) Markov predictor. The basic idea behind the predictor is that two pieces of information are used to generate the prediction, the phase id that was just seen, and the number of times prior to now that it has been seen in a row. The index into the prediction table is a hash of these two numbers.

additionally store the stable prediction. The table is updated by voting between the behavior just seen, the last phase id seen for this history (M1) and the stable phase id for this history (M2). If the outcome of this vote is different than the stable phase id, the stable phase id is updated to reflect this. The last phase id is then updated with the outcome.

We compare our phase prediction scheme with other prediction schemes in Figure 9. Figure 9, has four bars for every program with each bar corresponding to the prediction accuracy of a prediction scheme. The first and simplest scheme, `last` simply predicts that the phase identifier that will occur next is the same as the one just been observed, in essence always predicting stable operation. The prediction accuracy of this scheme is inversely proportional to the rate at which phases change in a given benchmark. For the program `vpr` for example, there is almost no change in phase and therefore predicting no-change does exceptionally well.

In order to counteract the effect that noise has on the simple predict-stable style of scheme, we further examined `vote3`. This works by predicting that the next phase will be the most commonly occurring of the past 3 phases seen. This removes many misses from `mesa` and `gzip`, but actually hurts for `bzip`.

Additionally we wanted to examine the effect of a simple Markov model based predictor. While this is very effective for `bzip`, it actually performs worse for `gzip` and `mesa`. We examined higher order Markov models but very little was gained in prediction accuracy for the additional area that was required to operate them.

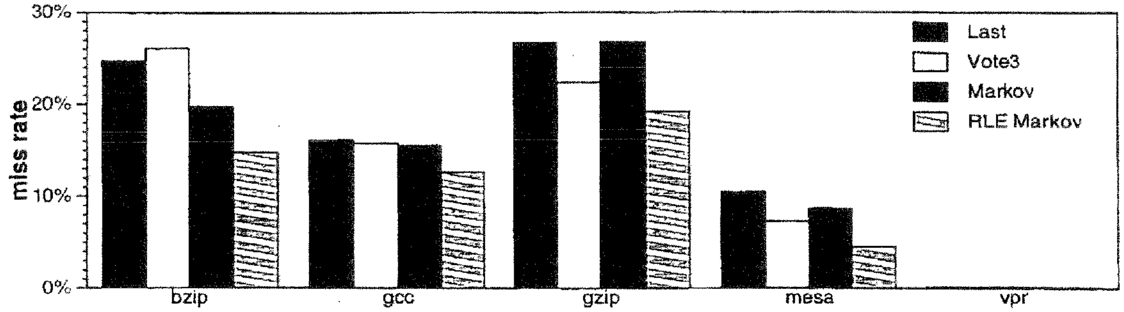


Figure 9: Phase ID Prediction Accuracy. This figure shows how well different prediction schemes work. The most naive scheme, `last`, simply predicts that the phases never change, while `vote3`, adds some noise reduction. The bars marked `Markov` and `RLE Markov` show how well we can predict the phase identifiers if we use a more sophisticated prediction scheme.

The final bar, `RLE Markov`, is our improved Markov predictor that is capable of compressing multiple periods in a particular phase into a tuple of phase id and duration. Using this scheme along with the hysteresis explained above, outperforms all other schemes on all the benchmarks we tested. It even provides some amount of benefit for `gcc` on which no other scheme provided any significant reduction in prediction accuracy.

7 Summary

In this paper we present an efficient run-time phase tracking architecture that is based on detecting changes in proportions of code executed. This is accomplished by dividing up all instructions seen into a set of buckets based on branch PCs. By doing this we approximate the effect of taking a random projection of the basic block vector, which was shown in [17], to be an effective method of compression. After an accumulation phase, where the instructions are tracked, the results are reduced to an even smaller vector for hardware phase classification.

The classification process compares the just gathered phase information with representatives from past phases, and attempts to find a close match. If a close match is found, the phases are identified as having the same phase id, and the phase ID can be used for prediction or statistic logging. If there is no match found, then it is marked as being a new phase and it is stored into the past footprint table to represent this newly found phase. We show that for most of the programs our phase tracker accurately identifies 90% or more of the phase transitions that occurred during execution for the metrics we examined (IPC, energy, and cache miss rates), while having less than 10% false positives.

In using this scheme for phase classification, we show that for most programs, a significant amount of the program (over 80%) is covered by 10 or less distinct phases. Furthermore, we show that these phases, while being distinct from one another, have fairly uniform behavior within a phase, meaning that most optimizations applied to one phase will work equally well on the other phases. In the program `gcc` the IPC attained by the processor on average over the full run of execution is 1.32, but has a standard deviation of more that 43%. By dividing it up into different phases, we get much more stable behavior, with IPCs ranging between 0.61 and 1.95 but now with standard deviations of less than 2%.

In addition to this we present a novel phase prediction architecture that can predict not only when a phase change is about to occur, but to which phase it is about to transition. The predictor uses the current phase id, along with the continuous duration of it's execution as a state from which to predict the next state. In using this scheme, we achieve average miss rates under 13% for the 4 programs (`gzip`, `bzip`, `gcc`, and `mesa`) that have interesting behavior.

We believe that use of our prediction classification architecture will lead to new optimizations in multi-threaded architecture scheduling, power management, and other resource distribution problems that must be controlled by the operating system but have effects at the micro-architectural level.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[2] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *International Symposium on High-Performance Computer Architecture (HPCA-7)*, January 2001.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *In Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED)*, July 2000.

[5] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[6] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[7] K. Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture*, pages 26–37, June 1997.

[8] Dirk Grunwald, Philip Levis, Charles B. Morrey III, Michael Neufeld, and Keith I. Farkas. Policies for dynamic clock scheduling. In *In Proceedings of the 2000 Conference on Operating Systems Design and Implementation*, December 2000.

[9] T. Heil and J. Smith. Relational profiling: Enabling thread level parallelism in virtual machines. In *In Proc. 33rd International Symposium on Microarchitecture*, December 2000.

[10] Michael Huang, Jose Renau, and Josep Torrellas. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[11] A. Klaiber. The technology behind crusoe processors. Technical report, Transmeta, January 2000.

[12] M. Martonosi, D. Clark, and M. Mesarina. The shrimp hardware performance monitor: Design and applications. 1996.

[13] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[14] S. Sastry, R. Bodik, and J.E. Smith. Rapid profiling via stratified sampling. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[15] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[18] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *In Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[19] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[20] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive mode control: A static-power-efficient cache design. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[21] C. Zilles and G. Sohi. A programmable co-processor for profiling. In *In Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001.

I hereby certify that this paper is being deposited with the United States Postal Service as Express Mail in an envelope addressed to: Box Provisional Application, Asst. Comm. for Patents, Washington, D.C. 20231, on this date.

Sep. 9, 2002
Date

Express Mail Label No.: *EV032727917US*

METHOD AND PROCESS TO CLASSIFY SIMILARITY AND PHASE-BASED BEHAVIOR IN COMPUTING SYSTEMS AND METHOD AND PROCESS FOR FINDING INTERVALS OF EXECUTION THAT ARE REPRESENTATIVE OF FULL EXECUTION

Modern computing systems are very complex and their behavior indicates this. Over time the system may go through a series of different "phases" over which time the behavior is fairly stable. This can happen at a variety of time scales, from nanoseconds to seconds. These phases can reoccur in the program, where the behavior of the program is similar in two or more related, but not necessarily contiguous, phases. The "behavior" of a phase is a general term used to describe any of variety of metrics which which are either directly measurable or inferable, and can include (but is not limited to) the number of hardware events (such as cache misses or branch mispredictions), the code executed, or power consumption rate. The problem we address is how to "accurately and efficiently" find, identify and classify similarity and phase-based behavior for software. The behavior we are interested in ranges from the behavior of the hardware, operating system, run-time system, and the software itself while executing the software.

Our approach finds similar sections of software execution regardless of temporal adjacency. In addition, it identifies when there is a phase change in behavior. The innovation of our technique is the ability to accurately and efficiently identify similarity and phase-based behavior while only examining the executed code for the software. The idea is based upon the observation that variance in program's behavior is caused by different parts of a program being executed. Therefore, to classify this variance, all we need to do is observe the behavior of the application based on the code being executed.

The similarity and phase behavior we find at the code level show highly correlated similarity and phase-based behavior for hardware, operating system and run-time system metrics. We are therefore able find similarity and phase-based behavior of hardware metrics, operating system performance, and run-time systems by only examining the software's executed code.

The two advantages of our approach (performing the analysis at the code level) are time and generality.

Depending upon the accuracy desired, our code level approach can analyze a program with only a small slowdown over the original program's execution. In comparison, performing this same analysis for a specific hardware, operating system, or run-time metric can slow down the system being monitored by 1000s of times.

The second advantage of our approach is that our analysis is generic. The similarity and phases we find can be used to represent similarity and phases found for executing that same program in terms of hardware, operating system, or run-time metrics. As an example, our similarity and phase-based analysis performed at the code level can accurately find similarity and phase-based behavior for the hardware metrics the software is being run upon (e.g., IPC, cache miss rates, power usage, frequency of JIT operations), software behavior (e.g., size and frequency of memory allocations, call stack depth), and operating system behavior (e.g., number of page faults, context switches), and potentially other types of behavior not listed here.

This patent application covers the following unique claims/ideas:

(1) Using a combination of the program structure and run time or simulation based analysis, to quickly find phase based behavior. This can be accomplished at the procedure level, basic block level, branch level, or by examining groups of instruction PCs executed close to each other in time.

(2) Using the similarity and phase-based behavior found via program structure for analysis or optimization of software, hardware, OS, or run-time system that in prior work required metric specific analysis. The benefit being that our program level analysis can be performed in a fraction of the time as the metric specific analysis.

(3) The use of (A) difference graphs, (B) signal theory, and (C) clustering to accurately classify similar execution and phase-based behavior.

(4) Efficient techniques for dealing with large data sets. We use random projections to map the large number of dimensions we deal with to a small number of dimensions without loosing accuracy in our analysis.

Timothy Sherwood and Brad Calder, Time Varying Behavior of Programs, UC San Diego Technical Report UCSD-CS99-630, August 1999 http://www-cse.ucsd.edu/~calder/papers/UCSD-CS99-630.pdf.

http://charlotte.ucsd.edu/~calder/simpoint/

With the speed in which we can perform similarity and phase-based software analysis there are many potential commercial applications.

Listed below are only a few applications:

(1) Finding Representative Simulation Points.

(2) Estimating and Predicting Machine Performance.

(3) Static and Dynamic Phase-based Software and Hardware Optimizations

(4) Profile based analysis tools to examine similarity and phase-based software behavior. A profiler could then be used by users to better tune their code and to take advantage of the behavior found.

The main approach for finding similarity and phase-based software behavior is to obtain this information using detail simulation. Our paper below was one of the first works to examine this at a general level using detailed simulation.:

Timothy Sherwood and Brad Calder, Time Varying Behavior of Programs, UC San Diego Technical Report UCSD-CS99-630, August 1999 http://www-cse.ucsd.edu/~calder/papers/UCSD-CS99-630.pdf

The goal of this invention is to automatically determine a small set of intervals of execution, we call Execution Points, that can be used to model the complete execution of the program. For these points to be useful, combined they must accurately model complete execution of the program at the hardware, operating system run-time and software level. These representative points can then be used to accurately model the complete execution of a program. 'Prior techniques for finding Execution Points rely upon sampling, which randomly takes points of execution until the probability of error goes below a desired threshold. This analysis is performed using detailed architecture and system level simulations.

In contrast, our approach uses our clustering analysis described in disclosure "Method and Process to Classify Similarity and Phase-based Behavior in Computing Systems" to quickly group the behavior of an application into its similar components. Since our approach is based upon only examining the behavior of the code at a high level, our initial clustering takes only a fraction of the time that prior sampling techniques take.

A goal of our approach is to find the minimal representative execution points to achieve a given level of accuracy. This is achieved by choosing one execution point from each cluster, and using the weight of the cluster to represent the execution weight for each point.

Since the clusters where intelligently formed, these minimal collection of execution points will together create an accurate representation of the program's complete execution. In comparison, sampling will require many execution points to achieve the same level of accuracy.

The unique claims/ideas in this disclosure cover the approaches we used to:

(1) Automatically finding the end of initialization for a program

(2) Automatically finding a minimized set of representative execution points to achieve a given level of accuracy, and the corresponding execution point weights

http://charlotte.ucsd.edu/~calder/simpoint/

no "lietery"

no "core"

no

Automatically Characterizing Large Scale Program Behavior

Timothy Sherwood Erez Perelman Greg Hamerly Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,eperelma,ghamerly,calder}@cs.ucsd.edu

Abstract

Understanding program behavior is at the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the very largest of scales (over the complete execution of the program). This realization has ramifications for many architectural and compiler techniques, from thread scheduling, to feedback directed optimizations, to the way programs are simulated. However, in order to take advantage of time-varying behavior, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

Our goal is to develop automatic techniques that are capable of finding and exploiting the Large Scale Behavior *of programs (behavior seen over billions of instructions). The first step towards this goal is the development of a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. To this end we examine the use of* Basic Block Vectors. *We quantify the effectiveness of Basic Block Vectors in capturing program behavior across several different architectural metrics, explore the large scale behavior of several programs, and develop a set of algorithms based on clustering capable of analyzing this behavior. We then demonstrate an application of this technology to automatically determine where to simulate for a program to help guide computer architecture research.*

1. INTRODUCTION

Programs can have wildly different behavior over their run time, and these behaviors can be seen even on the largest of scales. Understanding these large scale program behaviors can unlock many new optimizations. These range from new thread scheduling algorithms that make use of information on when a thread's behavior changes, to feedback directed optimizations targeted at not only the aggregate performance of the code but individual phases of execution, to creating simulations that accurately model full program behavior. To enable these optimizations, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

In order to perform such an analysis we need to develop a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. In [19], we presented the use of *Basic Block Vectors* (BBV), which uses the structure of the program that is exercised during execution to determine where to simulate. A BBV represents the code blocks executed during a given interval of execution. Our goal was to find a single continuous window of executed instructions that match the whole program's execution, so that this smaller window of execution can be used for simulation instead of executing the program to completion. Using the BBVs provided us with a hardware independent way of finding this small representative window.

In this paper we examine the use of BBVs for analyzing large scale program behavior. We use BBVs to explore the large scale behavior of several programs and discover the ways in which common patterns, and code, repeat themselves over the course of execution. We quantify the effectiveness of basic block vectors in capturing this program behavior across several different architectural metrics (such as IPC, branch, and cache miss rates).

In addition to this, there is a need for a way of classifying these repeating patterns so that this information can be used for optimization. We show that this problem of classifying sections of execution is related to the problem of *clustering* from machine learning, and we develop an algorithm to quickly and effectively find these sections based on clustering. Our techniques automatically break the full execution of the program up into several sets, where the elements of each set are very similar. Once this classification is completed, analysis and optimization can be performed on a per-set basis.

We demonstrate an application of this cluster-based behavior analysis to simulation methodology for computer architecture research. By making use of clustering information we are able to accurately capture the behavior of a whole program by taking simulation results from representatives of each cluster and weighing them appropriately. This results in finding a set of simulation points that when combined accurately represents the target application and input, which in turn allows the behavior of even very complicated programs such as gcc to be captured with a small amount of simulation time. We provide simulation points (points in the program to start execution at) for Alpha binaries of all of the SPEC 2000 programs. In addition, we validate these simulation points with the IPC, branch, and cache miss rates found for complete execution of the SPEC 2000 programs.

The rest of the paper is laid out as follows. First, a summary of the methodology used in this research is described

ASPLOS X, 10/02, San Jose, CA, USA.
Copyright 2002 ACM 1-58113-574-2/02/0010...$5.00.

in Section 2. Section 3 presents a brief review of basic block vectors and an in depth look into the proposed techniques and algorithms for identifying large scale program behaviors, and an analysis of their use on several programs. Section 4 describes how clustering can be used to analyze program behavior, and describes the clustering methods used in detail. Section 5 examines the use of the techniques presented in Sections 3 and 4 on an example problem: finding where to simulate in a program to achieve results representative of full program behavior. Related work is discussed in Section 6, and the techniques presented are summarized in Section 7.

2. METHODOLOGY

In this paper we used both ATOM [21] and SimpleScalar 3.0c [3] to perform our analysis and gather our results for the Alpha AXP ISA. ATOM is used to quickly gather profiling information about the code executed for a program. SimpleScalar is used to validate the phase behavior we found when clustering our basic block profiles showing that this corresponds to the phase behavior in the programs performance and architecture metrics. The baseline microarchitecture model we simulated is detailed in Table 1. We simulate an aggressive 8-way dynamically scheduled microprocessor with a two level cache design. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

We analyze and simulated all of the SPEC 2000 benchmarks compiled for the Alpha ISA. The binaries we used in this study and how they were compiled can be found at: `http://www.simplescalar.com/`.

3. USING BASIC BLOCK VECTORS

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies with which basic blocks are executed as the metric to compare different sections of the application's execution to one another. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block distributions provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides us with a fingerprint for that interval of execution, and tells us where in the code the application is spending its time. The basic idea is that knowing the basic block distribution for two different intervals gives us two separate fingerprints which we can then compare to find out how similar the intervals are to one another. If the fingerprints are similar, then the two intervals spend about the same amount of time in the same code, and the performance of those two intervals should be similar.

3.1 Basic Block Vector

A *Basic Block Vector* (BBV) is a single dimensional array, where there is a single element in the array for each static basic block in the program. For the results in this paper, the basic block vectors are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next interval of 100 million instructions. Therefore, each element in the array is the count of how many times the corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. By multiplying in the number of instructions in each basic block we insure that we weigh instructions the same regardless of whether they reside in a large or small basic block. We say that a Basic Block Vector which was gathered by counting basic block executions over an interval of $N \times 100$ million instructions, is a Basic Block Vector of duration N.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the proportions between time spent in basic blocks, a BBV is normalized by having each element divided by the sum of all the elements in the vector.

3.2 Basic Block Vector Difference

In order to find patterns in the program we must first have some way of comparing two Basic Block Vectors. The operation we desire takes as input two Basic Block Vectors, and outputs a single number which tells us how close they are to each other. There are several ways of comparing two vectors to one another, such as taking the dot product or finding the Euclidean or Manhattan distance. In this paper we use both the Euclidean and Manhattan distances for comparing vectors.

The Euclidean distance can be found by treating each vector as a single point in D-dimensional space. The distance between two points is simply the square root of the sum of squares just as in $c^2 = a^2+b^2$. The formula for computing the Euclidean distance of two vectors a and b in D-dimensional space is given by:

$$EuclideanDist(a, b) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

The Manhattan distance on the other hand is the distance between two points if the only paths you can take are parallel to the axes. In two dimensions this is analogous to the distance traveled if you were to go by car through city blocks. This has the advantage that it weighs more heavily differences in each dimension (being closer in the x-dimension does not get you any closer in the y-dimension). The Manhattan distance is computed by summing the absolute value of the element-wise subtraction of two vectors. For vectors a and b in D-dimensional space, the distance can be computed as:

$$ManhattanDist(a, b) = \sum_{i=1}^{D} |a_i - b_i|$$

Because we have normalized all of the vectors, the Manhattan distance will always be a single number between 0 and 2 (because we normalize each BBV to sum to 1). This number can then be used to compare how closely related two intervals of execution are to one another. For the rest of this section we will be discussing distances in terms of Manhattan distance, because we found that it more accurately represented differences in our high-dimensional data. We present the Euclidean distance as it pertains to the clustering algorithms presented in Section 4, since it provides a more accurate representation for data with lower dimensions.

3.3 Basic Block Similarity Matrix

Now that we have a method of comparing intervals of program execution to one another, we can now concentrate on finding phase-based behavior. A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being a contiguous interval of exe-

Instruction Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

cution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency.

A key observation from this paper is that the phase behavior seen in any program metric is directly a function of the code being executed. Because of this we can use the comparison between the Basic Block Vectors as an approximate bound on how closely related any other metrics will be between those two intervals.

To find how intervals of execution relate to one another we create a *Basic Block Similarity Matrix*. The similarity matrix is an upper triangular $N \times N$ matrix, where N is the number of intervals in the program's execution. An entry at (x, y) in the matrix represents the Manhattan distance between the basic block vector at interval x and the basic block vector at interval y.

Figures 1(left and right) and 4(left) shows the similarity matrices for gzip, bzip, and gcc using the Manhattan distance. The diagonal of the matrix represents the program's execution over time from start to completion. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter they are the more different they are (the Manhattan distance is closer to 2).

The top left corner of each graph is the start of program execution and is the origin of the graph, $(0, 0)$, and the bottom right of the graph is the point $(N-1, N-1)$ where N is the number of intervals that the full program execution was divided up into. The way to interpret the graph is to start considering points along the diagonal axis drawn. Each point is perfectly similar to itself, so the points directly on the axis all are drawn dark. Starting from a given point on the diagonal axis of the graph, you can begin to compare how that point relates to it's neighbors forward and backward in execution by tracing horizontally or vertically. If you wish to compare a given interval x with the interval at $x + n$, you simply start at the point (x, x) on the graph and trace horizontally to the right until you reach $(x, x + n)$.

To examine the phase behavior of programs, let us first examine gzip because it has behavior on such a large scale that it is easy to see. If we examine an interval taken from 70 billion instructions into execution, in Figure 1 (left), this is directly in the middle of a large phase shown by the triangle block of dark color that surrounds this point. This means that this interval is very similar to it's neighbors both forward and backward in time. We can also see that the execution at 50 billion and 90 billion instructions is also very similar to the program behavior at 70 billion. We also note, while it may be hard to see in a printed version that the phase interval at 70 billion instructions is similar to the phases at interval 10 and 30 billion, but they are not as similar as to those around 50 and 90 billion. Compare this with the IPC and data cache miss rates for gzip shown in Figure 2. Overall, Figure 1(left) shows that the phase behavior seen in the similarity matrix lines up quite closely with the behavior of the program, with 5 large phases (the first 2 being different from the last 3) each divided by a small phase, where all of the small phases are very similar to each other.

The similarity matrix for bzip (shown on the right of Figure 1) is very interesting. Bzip has complicated behavior, with two large parts to it's execution, compression and decompression. This can readily be seen in the figure as the large dark triangular and square patches. The interesting thing about bzip is that even within each of these sections of execution there is complex behavior. This, as will be shown later, makes the behavior of bzip impossible to capture using a small contiguous section of execution.

A more complex case for finding phase behavior is gcc, which is shown on the left of Figure 4. This similarity matrix shows the results for gcc using the Manhattan distance. The similarity matrix on the right will be explained in more detail in Section 4.2.1. This figure shows that gcc does have some regular behavior. It shows that, even here, there is common code shared between sections of execution, such as the intervals around 13 billion and 36 billion. In fact the strong dark diagonal line cutting through the matrix indicates that there is good amount of repetition between offset segments of execution. By analyzing the graph we can see that interval x is very similar to interval $(x + 23.6B)$ for all x.

Figures 2 and 5 show the time varying behavior of gzip and gcc. The average IPC and data cache miss rate is shown for each 100 million interval of execution over the complete execution of the program. The time varying results graphically show the same phase behavior seen by looking at only the code executed. For example, the two phases for gcc at 13 billion and 36 billion, shown to be very similar in Figure 4, are shown to have the same IPC and data cache miss rate in Figure 5.

4. CLUSTERING

The basic block vectors provide a compact and representative summary of the program's behavior for intervals of execution. By examining the similarity between them, it is clear that there exists a high level pattern to each program's execution. In order to make use of this behavior we need to start by delineating a method of finding and representing the information. Because there are so many intervals of execution that are similar to one another, one efficient representation is to group the intervals together that have similar behavior. This problem is analogous to a *clustering* problem. Later, in Section 5, we demonstrate how we use the clusters we discover to find multiple simulation points for irregular programs or inputs like gcc. By simulating only a single representative from each cluster, we can accurately represent the whole program's execution.

4.1 Clustering Overview

The goal of clustering is to divide a set of points into groups

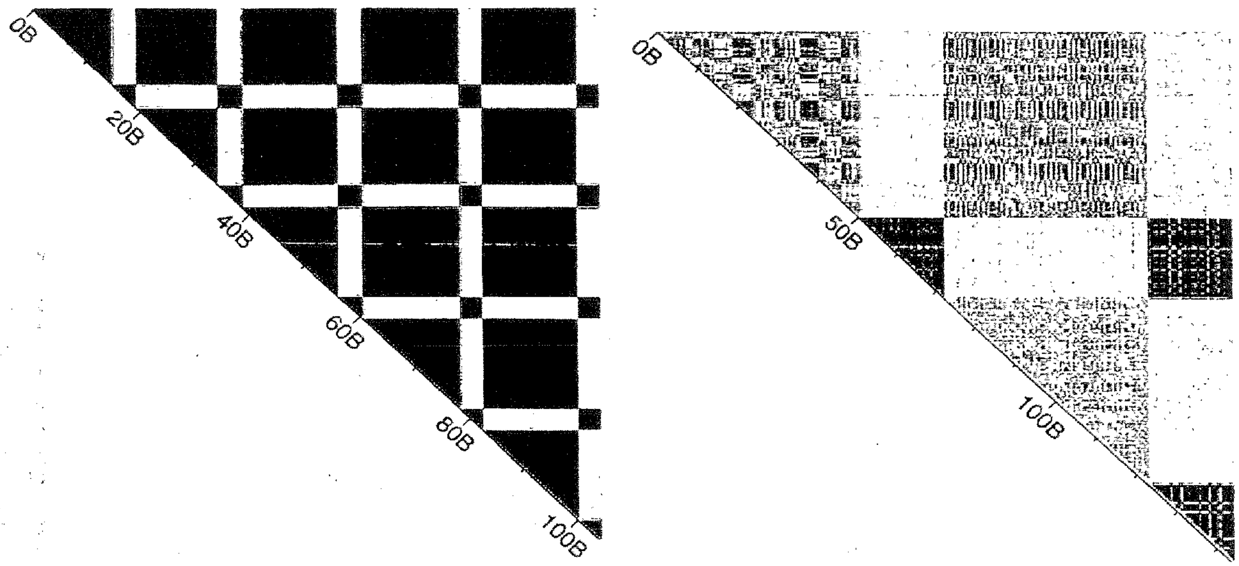
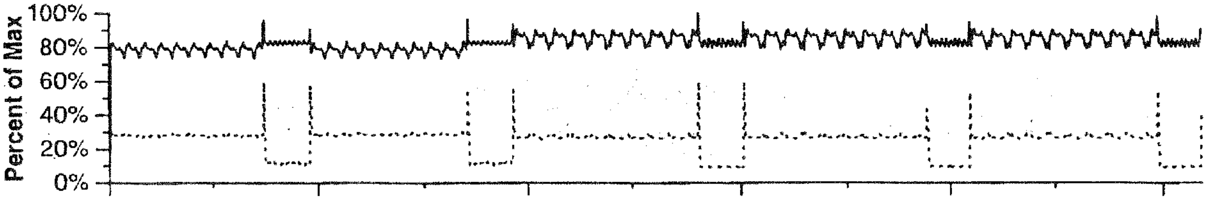


Figure 1: Basic block similarity matrix for the programs gzip-graphic (shown left) and bzip-graphic (shown right). The diagonal of the matrix represents the program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter the points the more different they are (the Manhattan distance is closer to 2).



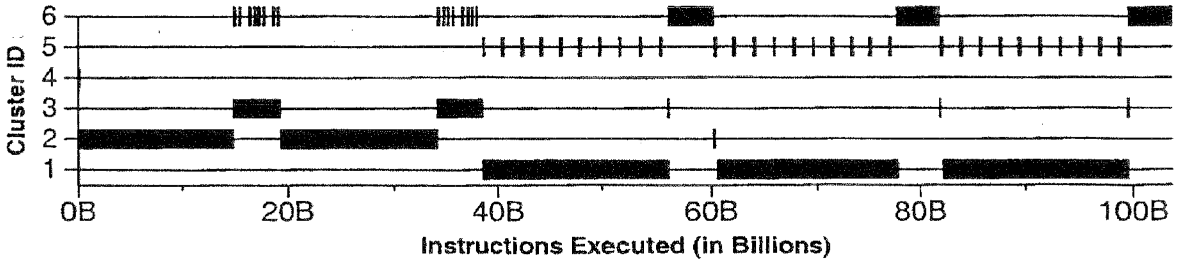


Figure 2: (top graph) Time varying graph for gzip-graphic. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 3: (bottom graph) Cluster graph for gzip-graphic. The full run of the execution is partitioned into a set of 6 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

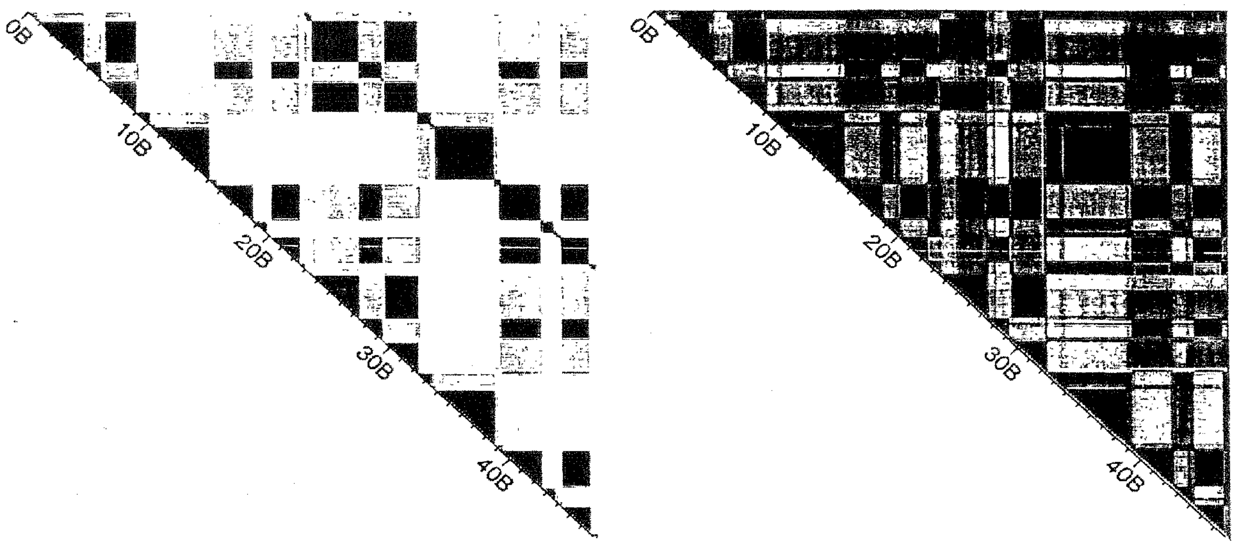
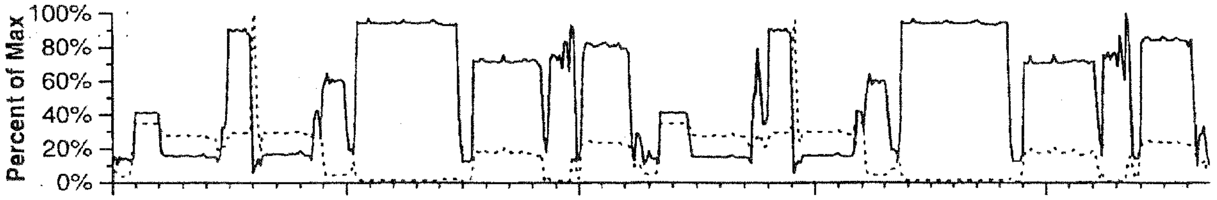


Figure 4: The original basic block similarity matrix for the program gcc (shown left), and the similarity matrix for gcc-166 drawn from projected data (on right). The figure on the left use the original basic block vectors (each of which has over 100,000 dimensions) and uses the Manhattan distance as a method of difference taking. The figure on the right uses projected data (down to 15 dimensions) and uses the Euclidean distance for difference taking.



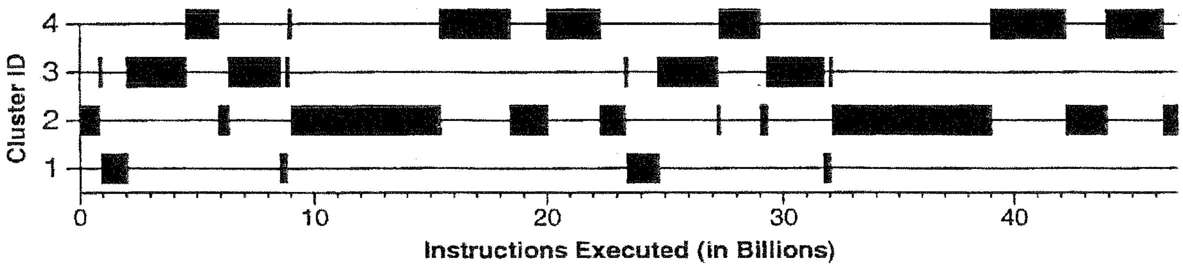


Figure 5: (top graph) Time varying graph for gcc-166. The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure 6: (bottom graph) Cluster graph for gcc-166. The full run of the execution is partitioned into a set of 4 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

such that points within each group are similar to one another (by some metric, often distance), and points in different groups are different from one another. This problem arises in other fields such as computer vision [10], document classification [22], and genomics [1], and as such it is an area of much active research. There are many clustering algorithms and many approaches to clustering. Classically, the two primary clustering approaches are Partitioning and Hierarchical:

Partitioning algorithms choose an initial solution and then use iterative updates to find a better solution. Popular algorithms such as k-means [14] and Gaussian Expectation-Maximization [2, pages 59–73] are in this family. These algorithms tend to have run time that is linear in the size of the dataset.

Hierarchical algorithms [9] either combine together similar points (called agglomerative clustering, and conceptually similar to Huffman encoding), or recursively divides the dataset into more groups (called divisive clustering). These algorithms tend to have run time that is quadratic in the size of the dataset.

4.2 Phase Finding Algorithm

For our algorithm, we use random linear projection followed by k-means. We choose to use the k-means clustering algorithm, since it is a very fast and simple algorithm that yields good results. To choose the value of k, we use the Bayesian Information Criterion (BIC) score [11, 17]. The following steps summarize our algorithm, and then several of the steps are explained in more detail:

1. Profile the basic blocks executed in each program to generate the basic block vectors for every 100 million instructions of execution.

2. Reduce the dimension of the BBV data to 15 dimensions using random linear projection.

3. Try the k-means clustering algorithm on the low-dimensional data for k values 1 to 10. Each run of k-means produces a clustering, which is a partition of the data into k different clusters.

4. For each clustering ($k = 1 \dots 10$), score the fit of the clustering using the BIC. Choose the clustering with the smallest k, such that it's score is at least 90% as good as the best score.

4.2.1 *Random Projection*

For this clustering problem, we have to address the problem of dimensionality. All clustering algorithms suffer from the so-called "curse of dimensionality", which refers to the fact that it becomes extremely hard to cluster data as the number of dimensions increases. For the basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for our experimental data, and could grow into the millions for very large programs. Another practical problem is that the running time of our clustering algorithm depends on the dimension of the data, making it slow if the dimension grows too large.

Two ways of reducing the dimension of data are dimension selection and dimension reduction. Dimension selection simply removes all but a small number of the dimensions of the data, based on a measure of goodness of each dimension for describing the data. However, this throws away a lot of data in the dimensions which are ignored. Dimension reduction reduces the number of dimensions by creating a new lower-dimensional space and then projecting each data point into the new space (where the new space's dimensions are not directly related to the old space's dimensions). This is analogous to taking a picture of 3 dimensional data at a random angle and projecting it onto a screen of 2 dimensions.

For this work we choose to use *random linear projection* [5] to create a new low-dimensional space into which we project the data. This is a simple and fast technique that is very effective at reducing the number of dimensions while retaining the properties of the data. There are two steps to reducing a dataset X (which is a matrix of basic block vectors and is of size $N_{intervals} \times D_{numbb}$, where D_{numbb} is the number of basic blocks in the program) down to D_{new} dimensions using random linear projection:

- Create a $D_{numbb} \times D_{new}$ projection matrix M by choosing a random value for each matrix entry between -1 and 1.

- Multiply X times M to obtain the new lower-dimensional dataset X' which will be of size $N_{intervals} \times D_{new}$.

For clustering programs, we found that using $D_{new} = 15$ dimensions is sufficient to still differentiate the different phases of execution. Figure 7 shows why we chose to project the data down to 15 dimensions. The graph shows the number of dimensions on the x-axis. The y-axis represents the k value found to be best on average, when the programs were projected down to the number of dimensions indicated by the x-axis. The best k is determined by the k with the highest BIC score, which is discussed in Section 4.2.3. The y-axis is shown as a percent of the maximum k seen for each program so that the curve can be examined independent of the actual number of clusters found for each program. The results show that for 15 dimensions the number of clusters found begins to stabilize and only climbs slightly. Similar results were also found using a different method of finding k in [6].

The advantages of using linear projections are twofold. First, creating new vectors with a low dimension of 15 is extremely fast and can even be done at simulation time. Secondly, using only 15 dimensions speeds up the k-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original basic block vectors.

Figure 4 shows the similarity matrix for gcc on the left using original BBVs, whereas the similarity matrix on the right shows the same matrix but on the data that has been projected down to 15 dimensions. For the reduced dimension data we use the Euclidean distance to measure differences, rather than the Manhattan distance used on the full data. After the projection, some information will be blurred, but overall the phases of execution that are very similar with full dimensions can still be seen to have a strong similarity with only 15 dimensions.

4.2.2 *K-means*

The k-means algorithm is an iterative optimization algorithm, which executes as two phases, which are repeated to convergence. The algorithm begins with a random assignment of k different centers, and begins its iterative process. The iterations are required because of the recursive nature of the algorithm; the cluster centers define the cluster membership for each data point, but the data point memberships define the cluster centers. Each point in the data belongs to, and can be considered a member of, a single cluster.

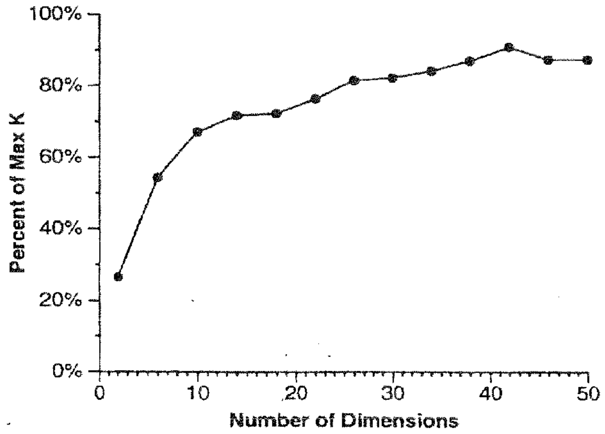


Figure 7: Motivation for random projection down to 15 dimensions (D=15). The x-axis is the number of dimensions of the projection, and the y-axis is the percent of the max number of clusters found for each program averaged over all spec programs. The results show that as you decrease the number of dimensions too far (the lowest point is two dimensions) the true clusters become collapsed on one another, and the algorithm cannot find as many clusters. By D=15 most of this effect has gone.

We initialize the k cluster centers by choosing k random points from the data to be clustered. After initialization, the k-means algorithm proceeds in two phases which are repeated until convergence:

- For each data point being clustered, compare its distance to each of the k cluster centers and assign it to (make it a member of) the cluster to which it is the closest.
- For each cluster center, change its position to the centroid of all of the points in its cluster (from the memberships just computed). The centroid is computed as the average of all the data points in the cluster.

This process is iterated until membership (and hence cluster centers) cease to change between iterations. At this point the algorithm terminates, and the output is a set of final cluster centers and a mapping of each point to the cluster that it belongs to. Since we have projected the data down to 15 dimensions, we can quickly generate the clusters for k-means with k from 1 to 10. In doing this, there are efficient algorithms for comparing the clusters that are formed for these different values of k, and choosing one that is good but still uses a small value for k is the next problem.

4.2.3 *Bayesian Information Criterion*

To compare and evaluate the different clusters formed for different k, we use the *Bayesian Information Criterion* (BIC) as a measure of the "goodness of fit" of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering being scored is a "good fit" to the data being clustered. We use the BIC formulation given in [17] for clustering with k-means, however other formulations of the BIC could also be used.

More formally, the BIC score is a penalized likelihood. There are two terms in the BIC: the likelihood and the penalty. The likelihood is a measure of how well the clustering models the data. To get the likelihood, each cluster is considered to be produced by a spherical Gaussian distribution, and the likelihood of the data in a cluster is the product of the probabilities of each point in the cluster given by the Gaussian. The likelihood for the whole dataset is just the product of the likelihoods for all clusters. However, the likelihood tends to increase without bound as more clusters are added. Therefore the second term is a penalty that offsets the likelihood growth based on the number of clusters. The BIC is formulated as

$$BIC(D,k) = l(D|k) - \frac{p_j}{2}\log(R)$$

where $l(D|k)$ is the likelihood, R is the number of points in the data, and p_j is the number of parameters to estimate, which is $(k-1)+dk+1$ for $(k-1)$ cluster probabilities, k cluster center estimates which each require d dimensions, and 1 variance estimate. To compute $l(D|k)$ we use

$$\begin{aligned} l(D|k) &= \sum_{i=1}^{k} -\frac{R_i}{2}\log(2\pi) - \frac{R_i d}{2}\log(\sigma^2) - \frac{R_i - 1}{2} \\ &\quad + R_i \log(R_i/R) \end{aligned}$$

where R_i is the number of points in the ith cluster, and σ^2 is the average variance of the Euclidean distance from each point to its cluster center.

For a given program and inputs, the BIC score is calculated for each k-means clustering, for k from 1 to N. We then choose the clustering that achieves a BIC score that is at least 90% of the spread between the largest and smallest BIC score that the algorithm has seen. Figure 8 shows the benefit of choosing a BIC with a high value and its relationship with the variance in IPC seen for that cluster. The y-axis shows the percent of IPC variance seen for a given clustering, and the corresponding BIC score the clustering received. Each point on the graph represents the average or max IPC variance for all points in the range of ±5% of the BIC score shown. The results show that picking clusterings that represent greater than 80% of the BIC score resulted in an IPC variance of less than 20% on average. The IPC variance was computed as the weighted sum of the IPC variance for each cluster, where the weight for a cluster is the number of points in that cluster. The IPC variance for each cluster is simply the variance of the IPC for all the points in that cluster.

4.3 Clusters and Phase Behavior

Figures 3 and 6 show the 6 clusters formed for gzip and the 4 clusters formed for gcc. The X-axis corresponds to the execution of the program in billions of instructions, and each interval (each of 100 million instructions) is tagged to be in one of the N clusters (labeled on the Y-axis). These figures, just as for Figures 1 and 4, show the execution of the programs to completion.

For gzip, the full run of the execution is partitioned into a set of 6 clusters. Looking to Figure 1(left) for comparison, we see that the cluster behavior captured by our tool lines up quite closely with the behavior of the program. The majority of the points are contained by clusters 1,2,3 and 6. Clusters 1 and 2 represent the large sections of execution which are similar to one another. Clusters 3 and 6 capture the smaller phases which lie in between these large phases, while cluster 5 contains a small subset of the larger phases, and cluster 4 represents the initialization phase.

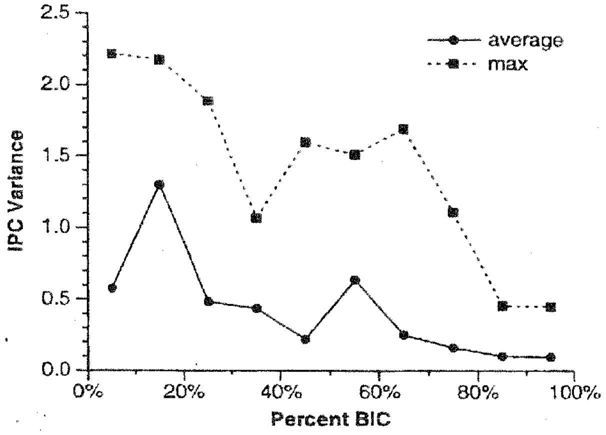


Figure 8: Plot of average IPC variance and max IPC variance versus the BIC. These results indicate that for our data, a clustering found to have a BIC score greater than 80% will have, on average, and IPC variance of less than 0.2.

In the cluster graph for gcc, shown in Figure 6, the run is now partitioned into 4 different clusters. Looking to Figure 4 for comparison, we see that even the more complicated behavior of gcc is captured correctly by our tool. Clusters 2 and 4 correspond to the dark boxes shown parallel to the diagonal axis. It should also be noted that the projection does introduce some degree of error into the clustering. For example, the first group of points in cluster 2 are not really that similar to the other points in the cluster. Comparing the two similarity matrices in Figure 4, shows the introduction of a dark band at (0,30) on the graph which was not in the original (un-projected) data. Despite these small errors, the clustering is still very good, and the impact of any such errors will be minimized in the next section.

5. FINDING SIMULATION POINTS

Modern computer architecture research relies heavily on cycle accurate simulation to help evaluate new architectural features. While the performance of processors continues to grow exponentially, the amount of complexity within a processor continues to grow at an even a faster rate. With each generation of processor more transistors are added, and more things are done in parallel on chip in a given cycle while at the same time cycle times continue to decrease. This growing gap between speed and complexity means that the time to simulate a constant amount of processor time is growing. It is already to the point that executing programs fully to completion in a detailed simulator is no longer feasible for architectural studies. Since detailed simulation takes a great deal of processing power, only a small subset of a whole program can be simulated.

SimpleScalar [3], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time.

Because it is only feasible to execute a small portion of the program, it is very important that the section simulated is an accurate representation of the program's behavior as a whole. The basic block vector and cluster analysis presented in Sections 3 and 4 will allow us to make sure that this is the case.

5.1 Single Simulation Points

In [19], we used basic block vectors to automatically find a single simulation point to potentially represent the complete execution of a program. A *Simulation Point* is a starting simulation place (in number of instructions executed from the start of execution) in a program's execution derived from our analysis. That algorithm creates a target basic block vector, which is a BBV that represents the complete execution of the program. The Manhattan distance between each interval BBV and the target BBV is computed. The BBV with the lowest Manhattan distance represents the single simulation point that executes the code closest to the complete execution of the program. This approach is used to calculate the long single simulation points (LongSP) described below.

In comparison, the single simulation point results in this paper are calculated by choosing the BBV that has the smallest Euclidean distance from the centroid of the whole dataset in the 15-dimensional space, a method which we find superior to the original method. The 15-dimensional centroid is formed by taking the average of each dimension over all intervals in the cluster.

Figure 9 shows the IPC estimated by executing only a single interval, all 100 million instructions long but chosen by different methods, for all SPEC 2000 programs. This is shown in comparison to the IPC found by executing the program to completion. The results are from SimpleScalar using the architecture model described in Section 2, and all fast forwarding is done so that all of the architecture structures are completely warmed up when starting simulation (no cold-start effect).

The first bar, labeled `none`, is the IPC found when executing only the first 100 million instructions from the start of execution (without any fast forwarding). The second bar, `FF-Billion` shows the results after blindly fast forwarding 1 billion instructions before starting simulation. The third bar, `SimPoint` shows the IPC using our single simulation point analysis described above, and the last bar shows the IPC of simulating the program to completion (labeled `Full`). Because these are actual IPC values, values which are closer to the `Full` bar are better.

The results in Figure 9 shows that the single simulation points are very close to the actual full execution of the program, especially when compared against the ad-hoc techniques. Starting simulation at the start of the program results in an average error of 210%, whereas blindly fast forwarding results in an average 80% IPC error. Using our single simulation point analysis we reduce the average IPC error to 18%. These results show that it is possible to reasonably capture the behavior of the most programs using a very small slice of execution.

Table 2 shows the actual simulation points chosen along with the program counter (PC) and procedure name corresponding to the start of the interval. If an input is not attached to the program name, then the default ref input was used. Columns 2 through 4 are in terms of the number of intervals (each 100 million instruction long). The first column is the number of instructions executed by the program, on the specific input, when run to completion. The second column shows the end of initialization phase calculated as described in [19]. The third column shows the single simulation point automatically chosen as described above. This simu-

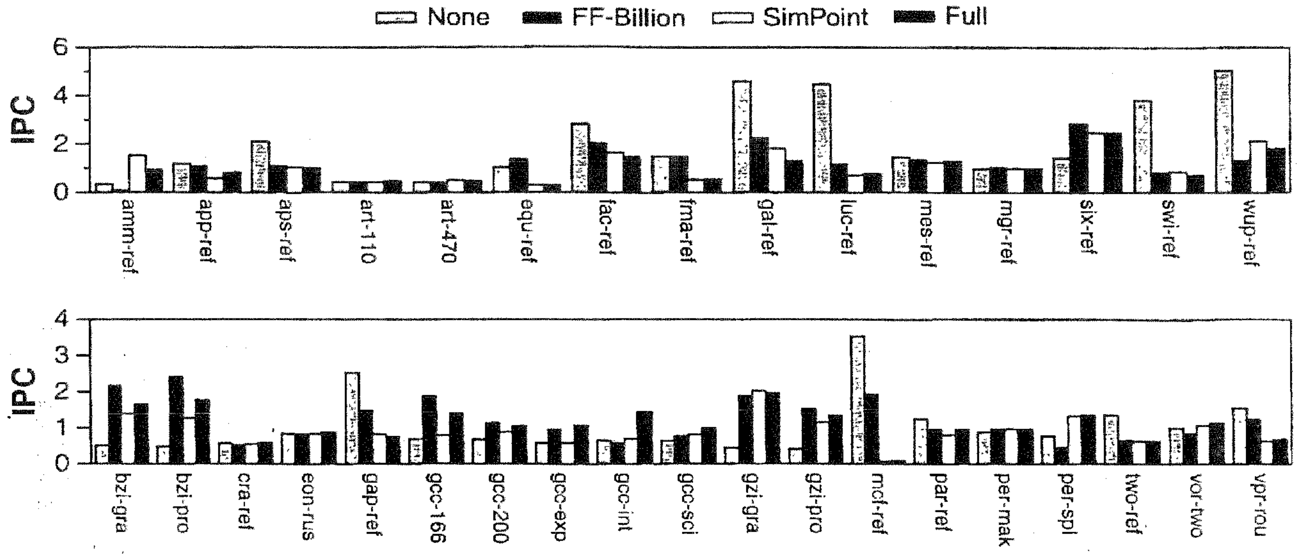


Figure 9: Simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

lation point is used to fast forward to the point of desired execution. Some simulators, debuggers, or tracing environments (e.g., gdb) provide the ability to fast forward based upon a program PC, and the number of times that PC was executed. We therefore, provide the instruction PC for the start of the simulation point, the procedure that PC occurred in, and the number of times that PC has to be executed in order to arrive at the desired simulation point.

These results show that a single simulation point can be accurate for many programs, but there is still a significant amount of error for programs like bzip, gzip and gcc. This occurs because there are many different phases of execution in these programs, and a single simulation point will not accurately represent all of the different phases. To address this, we used our clustering analysis to find multiple simulation points to accurately capture these programs behavior, which we describe next.

5.2 Multiple Simulation Points

To support multiple simulation points, the simulator can be run from start to stop, only performing detailed simulation on the selected intervals. Or the simulation can be broken down into N simulations, where N is the number of clusters found via analysis, and each simulation is run separately. This has the further benefit of breaking the simulation down into parallel components that can be distributed across many processors. This is the methodology we use in our simulator. For both cases results from the separate simulation points need to be weighed and combined to arrive at overall performance for the program [4]. Care must be taken to combine statistics correctly (simply averaging will give incorrect results for statistics such as rates).

Knowing the clustering alone is not sufficient to enable multiple point simulation because the cluster centers do not correspond to actual intervals of execution. Instead, we must first pick a representative for each cluster that will be used to approximate the behavior of the the full cluster. In order to pick this representative, we choose for each cluster the actual interval that is closest to the center (centroid) of the cluster. In addition to this, we weigh any use of this representative by the size of the cluster it is representing. If a cluster has only one point, it's representative will only have a small impact on the overall outcome of the program.

Table 2 shows the multiple simulation points found for all of the SPEC 2000 benchmarks. For these results we limited the number of clusters to be at most six for all but the most complex programs. This was done, in order to limit the number of simulation points, which also limits the amount of warmup time needed to perform the overall simulation. The cluster formation algorithm in Section 4 takes as an input parameter the max number of clusters to be allowed. Each simulation point contains two numbers. The first number is the location of the simulation point in 100s of millions of instructions. The second number in parentheses is the weight for that simulation point, which is used to create an overall combined metric. Each simulation point corresponds to 100 million instructions.

Figure 10 shows the IPC results for multiple simulation points. The first bar shows our single simulation points simulating for 100 million instructions. The second bar LongSP chooses a single simulation point, but the length of simulation is identical to the length used for multiple simulation points (which may go up to 1 billion instructions). This is to provide a fair comparison between the single simulation points and multiple. The Multiple bar shows results using the multiple simulation points, and the final bar is IPC for full simulation. As in Figure 9, the closer the bar is to Full, the better.

The results show that the average IPC error rate is reduced to 3% using multiple simulation points, which is down from 17% using the long single simulation point. This is significantly lower than the average 80% error seen for blindly fast forwarding. The benefits can be most clearly seen in the programs bzip, gcc, ammp, and galgel. The reason that the long contiguous simulation points do not do much better is that they are constrained to only sample at one place in the program. For many programs this is sufficient, but for those with interesting long term behavior, such as bzip, it is

name	Len	Init	SP	PC	Proc Name	Multiple SimPoints				

ammp	3265	24	109	026834	mm_fv_update.	3026(13.8)
						1607(12.6)
							1774(31)
							2437(4.9)
								595(15.3)
								3112(11.5)
									1068(1.3)
									2480(2.2)
										2128(7.4)

applu	2238	3	2180	018520	buts_	624(22.1)
						1507(14.5)
							1625(22.5)	1956(18.8)	2234(6.6)	1380(15.5)

apsi	3479	3	3409	0380ac	dctdxf_	2107(5.6)	2863(14)	1007(70.7)	896(7.7)	1618(2)
art-110	417	75	341	00fbb0	match	82(42.9)	255(41.2)	50(15.8)		
art-470	450	83	366	00f5d0	match	300(36.2)	46(14.7)	236(49.1)		

bzip2-graphic	1435	4	719	012a5c	spec_putc	168(11.7)
						519(11.6)
							1042(3.7)
							872(8.2)
								430(7.5)
								195(5.6)
									762(16.2)
									148(2)
										106(15.3)
										1435(18.2)

bzip2-program	1249	4	459	00ddd0	sortIt	140(11)
						1005(7)
							468(12.3)
							94(6.9)
								78(6.2)
								606(14)
									990(16)
									859(14.6)
										445(7.4)
										341(4.7)

bzip2-source	1088	4	978	00d774	qSort3	395(16)
						177(34.7)
							511(4.3)	64(29.1)	488(7.3)	530(8.6)

crafty	1918	462	775	021730	SwapXray	123(25)	510(19.7)	664(22.7)	1123(32.5)	
eon-rushmeier	578	140	404	04e1b4	viewingHit	260(6.6)	238(23.7)	337(20.9)	435(35.6)	216(13.1)

equake	1315	35	813	012410	phi0	874(12.2)
						62(11.6)
							1292(36.7)	463(12.2)	336(24.1)	3(3.2)

facerec	2682	356	376	02d1f4	graphroutines_lo.	1976(60.1)	1528(2.5)	1935(3.9)	1398(29.2)	348(4.3)

fma3d	2683	192	2542	0e3140	scatter_element.	112(7)
						509(13)
							209(0.6)	842(68.4)	1600(11)	47(0.1)

galgel	4093	3	2492	02db00	syshtn_	3511(5.5)
						2181(29)
							2081(11)
							2161(3.3)
								3466(11.2)
								1017(5.5)
									516(31.6)	2141(2.7)

gap	2695	639	675	050750	CollectGarb	1114(8.2)	1196(58.1)	88(12.7)	2189(14)	2609(7.1)
gcc-166	469	61	390	0d157c	gen_rtx	238(6.4)	149(42.2)	30(21.3)	404(30.1)	
gcc-200	1086	151	737	0ceb04	refers_to_regno.	8(45.8)	587(17.9)	921(10.9)	575(14.5)	1011(11)

gcc-expr	120	27	37	191fd0	validate_change	63(12.5)
						88(5)
							81(15.8)	42(16.7)	25(4.2)	9(45.8)

gcc-integrate	131	14	5	1198e0	find_single_use.	118(9.2)
						73(17.6)
							41(27.5)	102(21.4)	9(20.6)	57(3.8)

gcc-scilab	620	139	208	100d54	insert	255(54.2)	39(9.5)	231(13.2)	379(15.8)	170(7.3)

gzip-graphic	1037	158	654	009c00	fill_window	961(45.4)
						566(13.4)
							87(28.5)	373(7.3)	1(0.1)	461(5.2)

gzip-log	395	91	266	00d280	inflate_codes	207(24.1)	171(16.5)	157(16.7)	330(23.5)	71(19.2)
gzip-program	1688	112	1190	009660	longest_match	228(22.7)	779(21.4)	472(9.1)	1410(20.4)	594(26.4)

gzip-random	821	152	624	00a14c	deflate	484(0.9)
						1(0.1)
							625(0.2)	580(51)	811(16.8)	200(30.9)

gzip-source	843	68	335	00a224	deflate	248(14.5)
						720(2.5)
							327(13.2)	167(17.7)	656(27.8)	373(24.4)

lucas	1423	11	546	021ef0	fft_square_	982(21.4)	602(10.7)	1370(21.4)	458(28)	524(18.6)

mcf	618	15	554	00911c	price_out_impl k	268(39.6)
						143(3.9)
							425(11)	205(30.1)	468(4.5)	316(10.8)

mesa	2816	6	1136	0a30f0	general_textured.	1846(35.3)	2806(0.7)	398(35.3)	977(28.8)	
mgrid	4191	21	3293	0160f0	resid_	43(24.2)	3459(22.8)	807(20.1)	3110(16.3)	2476(16.6)
parser	5467	388	1147	01edfc	region_valid	3342(25.1)	1771(29.8)	5102(19.7)	2008(19.4)	4772(6)

perlbmk-diff	399	56	142	07f974	regmatch	6(1)
						239(31.8)
							355(62.7)	11(0.5)	397(0.8)	12(3.3)

perlbmk-make	20	3	12	08268c	Perl_runops_st.	1(5)	20(20)	6(75)		
perlbmk-perf	290	69	6	08268c	Perl_runops_st.	39(59.3)	207(40.7)			
perlbmk-split	1108	162	451	07fc98	regmatch	704(44.9)	596(9.1)	232(21.7)	461(21.8)	501(2.6)
sixtrack	4709	250	3044	167894	thin6d_	6(1.7)	1719(98.3)			
swim	2258	3	2080	019130	calc1_	1951(29.8)	38(14)	777(24.7)	710(13.8)	2101(17.8)
twolf	3464	7	1067	041094	ucxx1	312(17)	2888(11.3)	3268(11.7)	961(20.4)	2054(39.5)
vortex-one	1189	36	272	06289c	Mem_GetWord	536(17.1)	366(23.3)	115(8.2)	1068(17.2)	878(34.2)

vortex-three	1330	177	565	0336a8	Part_Delete	934(25.4)
						485(25.4)
							1129(11.4)	96(8.9)	47(11.1)	586(17.8)

vortex-two	1386	206	1025	05e6fc	Mem_NewRegion	635(7.6)
						397(23.2)
							752(24.5)	554(21.9)	930(7.4)	360(15.3)

vpr-place	1122	4	593	0224ec	get_non_update.	166(25.5)
						547(27.9)
							857(21.6)	1(0.2)	362(12.8)	1057(12)

vpr-route	840	12	477	025c80	get_heap_head	559(29.9)	89(28)	353(23.8)	3(2.6)	490(15.7)
wupwise	3496	11	3238	01d680	zgemm_	1811(43.3)	91(8)	3055(43.2)	1524(5.4)	

Table 2: Single simulation points for SPEC 2000 benchmarks. Columns 2 through 4 are in terms of 100 million instruction executed. The length of full execution is shown, as well as the end of initialization. SP is the single simulation point using the approach in this paper. The procedure in which the simulation point occurred and its PC are also shown. The last 6 digits of PC of each SimPoint is given in hex, so the address is formed from 120xxxxxx. Procedure names that end in "." were truncated due to space. The rest of the columns list the multiple simulation points found in 100s of millions. The first number is the starting place of the simulation point relative to the *start* of execution. The second number shows the weight given to the cluster that simulation point was taken from, and is used when weighing the final results of the simulation.

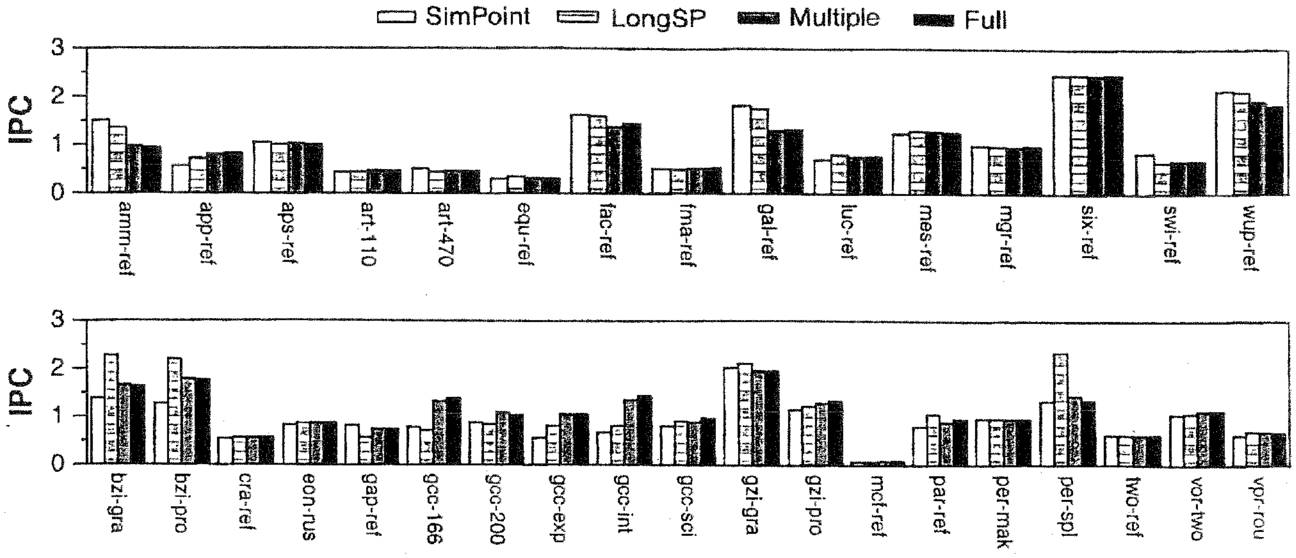


Figure 10: Multiple simulation point results. Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

impossible to approximate the full behavior.

Figure 11 is the average over all of the floating point programs (top graph) and integer programs (bottom graph). Errors for IPC, branch miss rate, instruction and data cache miss rates, and the unified L2 cache miss rate for the architecture presented in Section 2 are shown. The errors are with respect to these metrics for the full length of simulation using SimpleScalar. Results are shown for starting simulation at the start of the program `None`, blindly fast forwarding a billion instructions `FF-Billion`, single simulation points of duration 1 (`SimPoint`) and k (`LongSP`), and multiple simulation points (`Multiple`).

The first thing to note is that using the just a single small simulation point performs quite well on average across all of the metrics when compared to blindly fast-forwarding. Even though a single SimPoint does well, it is clearly beaten by using the clustering based scheme presented in this paper across all of the metrics examined. One thing that stands out on the graphs is that the error rate of the instruction cache and L2 cache appear to be high (especially for the integer programs) despite the fact that our technique is doing quite well in terms of overall performance. This is due to the fact that we present here an arithmetic mean of the errors, and there are several programs that have high error rates due to the very small number of cache misses. If there are 10 misses in the whole program, and we estimate there to be 100, that will result in a error of 10X. We point to the overall IPC as the most important metric for evaluation as it implicitly weighs each of the metrics by it's relative importance.

6. RELATED WORK

Time Varying Behavior of Programs: In [18], we provided a first attempt at showing the periodic patterns for all of the SPEC 95 programs, and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy.

Training Inputs and Finding Smaller Representative Inputs: One approach for reducing the simulation time is to use the training or test inputs from the SPEC benchmark suite. For many of the benchmarks, these inputs are either (1) still too long to fully simulate, or (2) too short and place too much emphasis on the startup and shutdown parts of the program's execution, or (3) inaccurately estimate behavior such as cache accesses do to decreased working set size.

KleinOsowski et. al [12], have developed a technique where they manually reduce the input sets of programs. The input sets were developed using a range of approaches from truncating of the input files to modification of source code to reduce the number of times frequent loops were traversed. For these input sets they develop, they make sure that they have similar results in terms of IPC, cache, and instruction mix.

Fast Forwarding and Check-pointing: Historically researchers have simulated from the start of the application, but this usually does not represent the majority of the program's behavior because it is still in the initialization phase. Recently researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. During fast-forward the simulator simply needs to act as a functional simulator, and may take full advantage of optimizations like direct execution. After the fast-forward point has been reached, the simulator switches to full cycle level simulation.

After fast-forwarding, the architecture state to be simulated is still cold, and a warmup time is needed in order to start collecting representative results. Efficiently warming up execution only requires references immediately proceeding the start of simulation. Haskins and Skadron [7] examined probabilistically determining the minimum set of fast-forward transactions that must be executed for warm up to accurately produce state as it would have appeared had the entire fast-forward interval been used for warm up [7]. They

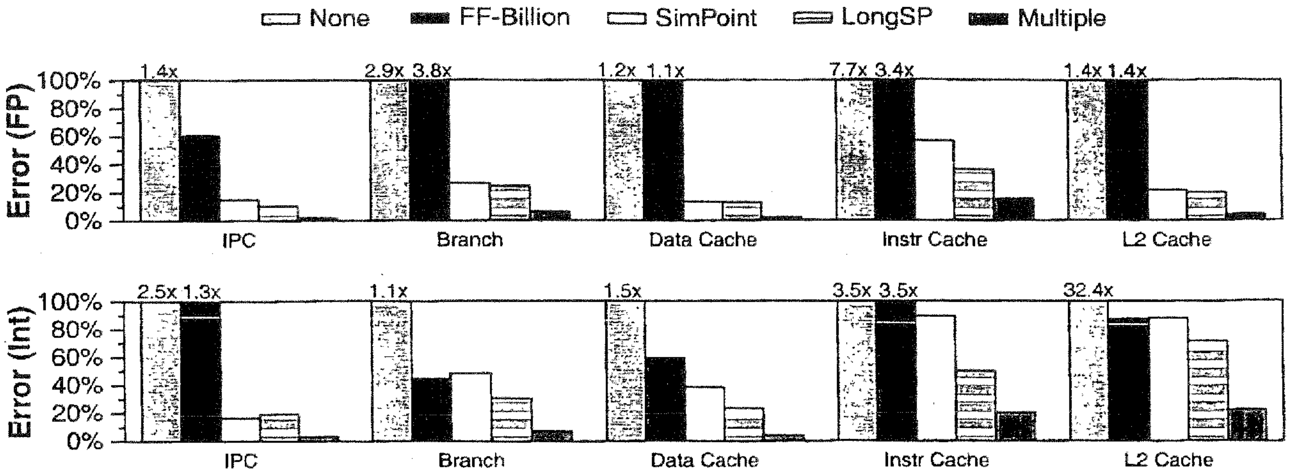


Figure 11: Average error results for the SPEC 2000 floating point (top) and integer (bottom) benchmarks for IPC, branch misprediction, instruction, data and unified L2 cache miss rates.

recently examined using reuse analysis to determine how far before full simulation warmup needs to occur [8].

An alternative to fast forwarding is to use check-pointing to start the simulation of a program at a specific point. With check-pointing, code is executed to a given point in the program and the state is saved, or checkpointed, so that other simulation runs can start there. In this way the initialization section can be run just one time, and there is no need to fast forward past it each time. The architectural state (e.g., caches, register file, branch prediction, etc) can either be stored in the trace (if they are not going to change across simulation runs) or can be warmed up in a manner similar to described above.

Automatically Finding Where to Simulate: Our work is based upon the basic block distribution analysis in [19] as described in prior sections. Recent work on finding simulation points for data cache simulations is presented by Lafage and Seznec [13]. They proposed a technique to gather statistics over the complete execution of the program and use them to choose a representative slice of the program. They evaluate two metrics, one which captures memory spatial locality and one which captures memory temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of the both the program's full execution and the execution of samples.

Statistical Sampling: Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These techniques take a number of contiguous execution samples, referred to as clusters in [4], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated. Conte et. al [4] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC and branch and data cache statistics). Our work is complementary to this, where we provide a fast and metric independent approach for picking multiple simulation points based just on basic block vector similarity. When an architect gets a new binary to examine they can use our approach to quickly find the simulation points, and then validate these with detailed simulation in parallel with using the binary.

Statistical Simulation: Another technique to improve simulation time is to use statistical simulation [16]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. Nussbaum and Smith [15] also examined generating synthetic traces and using these for simulation and was proposed for fast design space exploration. We believe the techniques presented in this paper are complementary to the techniques of Oskin et al. and Nussbaum and Smith in that more accurate profiles can be determined using our techniques, and instead of attempting to characterize the program as a whole it can be characterized on a per-phase basis.

7. SUMMARY

At the heart of computer architecture and program optimization is the need for understanding program behavior. As we have shown, many programs have wildly different behavior on even the very largest of scales (over the full lifetime of the program). While these changes in behavior are drastic, they are not without order, even in very complex applications such as gcc. In order to help future compiler and architecture researchers in exploiting this large scale behavior, we have developed a set of analytical tools that are capable of automatically and efficiently analyzing program behavior over large sections of execution.

The development of the analysis is founded on a hardware independent metric, *Basic Block Vectors*, that can concisely summarize the behavior of an arbitrary section of execution in a program. We showed that by using Basic Block Vectors one can capture the behavior of programs as defined by several architectural metrics (such as IPC, and branch and cache miss rates).

Using this framework, we examine the large scale behavior of several complex programs like gzip, bzip, and gcc, and find interesting patterns in their execution over time. The behavior that we find shows that code and program behavior repeat over time. For example, in the input we examined in detail for gcc we see that program behavior repeats itself every 23.6 billion instructions. Developing techniques that automatically capture behavior on this scale is useful for architectural, system level, and runtime optimizations. We present an algorithm based on the identification of clusters of basic block vectors that can find these repeating program behaviors and group them into sets for further analysis. For two of the programs gzip and gcc we show how the clustering algorithm results line up nicely with the similarity matrix and correlate with the time varying IPC and data cache miss rates.

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program. We show how our clustering analysis can be used to automatically find multiple simulation points to reduce simulation time and to accurately model full program behavior. We call this clustering tool to find single and multiple simulation points *SimPoint*. SimPoint along with additional simulation point data can be found at: `http://www.cs.ucsd.edu/~calder/simpoint/`. For the SPEC 2000 programs, we found that starting simulation at the start of the program results in an average error of 210% when compared to the full simulation of the program, whereas blindly fast forwarding resulted in an average 80% IPC error. Using a single simulation point found, using our basic block vector analysis, resulted in an average 17% IPC error. When using the clustering algorithm to create multiple simulation points we saw an average IPC error of 3%.

Automatically identifying the phase behavior using clustering is beneficial for architecture, compiler, and operating system optimizations. To this end, we have used the notion of basic block vectors and a random projection to create an efficient technique for identifying phases on-the-fly [20], which can be efficiently implemented in hardware or software. Besides identifying phases, this approach can predict not only when a phase change is about to occur, but to which phase it is about to transition. We believe that using phase information can lead to new compiler optimizations with code tailored to different phases of execution, multi-threaded architecture scheduling, power management, and other resource distribution problems controlled by software, hardware or the operating system.

Acknowledgments

We would like to thank Suleyman Sair and Chris Weaver for their assistance with SimpleScalar, as well as Mark Oskin and the anonymous reviewers for providing helpful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045 and NSF CAREER grant No. CCR-9733278.

8. REFERENCES

[1] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6:281–297, 1999.

[2] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[4] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[5] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[6] G. Hamerly and C. Elkan. Learning the k in k-means. Technical Report CS2002-0716, University of California, San Diego, 2002.

[7] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[8] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerating sampled microarchitecture simulations. Technical Report CS-2002-19, U of Virginia, July 2002.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[10] J.-M. Jolion, P. Meer, and S. Bataouche. Robust clustering with applications in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):791–802, 1991.

[11] R. E. Kass and L. Wasserman. A reference Bayesian test for nested hypotheses and its relationship to the schwarz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[12] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[13] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[15] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[16] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[17] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[18] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[19] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[20] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. Technical Report CS2002-0710, UC San Diego, June 2002.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[22] O. Zamir and O. Etzioni. Web document clustering: A feasibility demonstration. In *Research and Development in Information Retrieval*, pages 46–54, 1998.

In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT),

September 2001.

Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications

Timothy Sherwood Erez Perelman Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,eperelma,calder}@cs.ucsd.edu

Abstract

Modern architecture research relies heavily on detailed pipeline simulation. Simulating the full execution of an industry standard benchmark can take weeks to months to complete. To overcome this problem researchers choose a very small portion of a program's execution to evaluate their results, rather than simulating the entire program.

In this paper we propose Basic Block Distribution Analysis *as an automated approach for finding these small portions of the program to simulate that are representative of the entire program's execution. This approach is based upon using profiles of a program's code structure (basic blocks) to uniquely identify different phases of execution in the program. We show that the periodicity of the basic block frequency profile reflects the periodicity of detailed simulation across several different architectural metrics (e.g., IPC, branch miss rate, cache miss rate, value misprediction, address misprediction, and reorder buffer occupancy). Since basic block frequencies can be collected using very fast profiling tools, our approach provides a practical technique for finding the periodicity and simulation points in applications.*

1 Introduction

In order to evaluate new architecture features, detailed modeling of the pipeline, buses, and queuing delays are needed along with timing models and power estimation. Detailed simulation takes a great deal of processing power and time, and only a small subset of a whole program is often simulated. Many programs have wildly different behavior during different parts of their execution making the section of the program's execution simulated of great importance to the relevance and correctness of the study.

In [10], we found, when looking at architecture features, that most programs demonstrate cyclic behavior across many different metrics. These include IPC, branch prediction, value prediction, address prediction, cache performance, and reorder buffer occupancy. Cyclic (periodic) behavior of an application is defined as a repeatable pattern seen for the metric throughout the program's execution. For example, the SPEC95 program wave, shows two main phases to its cycle. It has an IPC of 3 during the 1st phase, and an IPC of 2 during the 2nd phase, and this repeats throughout its execution. The period is the length of time it takes to complete both phases of its cycle.

The main focus of our paper is to develop an automated, accurate, and efficient approach for determining the starting points in a program to simulate and the duration of the simulation. We focus on finding:

i. The end of the initialization part of the program, and the start of the cyclic part of the program.

ii. The period of the program. The period is the length of the cyclic nature found during a program's execution.

iii. The ideal place to simulate given a specific number of instructions one has time to simulate.

iv. An accurate confidence estimation of the simulation point.

To create a fast and efficient tool, we focused on an approach that does not use any knowledge of the architectural metrics for the program, but is instead highly correlated with the performance of those metrics.

We propose using *Basic Block Distribution Analysis* (BBDA) to calculate the above enumerated items. When running a program to completion, it will execute each basic block a certain number of times. Taking a snapshot of the number of times each basic block is executed provides us with a basic block fingerprint. We use basic block fingerprints gathered for small intervals of the program's execution to find representative areas of the program to simulate. This is done by finding the best match of these smaller basic block fingerprints to a basic block fingerprint representing the complete execution of the program.

A potential advantage of BBDA is that it only requires basic block profiles, which means a relatively fast basic block profiler is used (as opposed to slow timing simulation). In addition, many compilers already collect basic block and edge frequency profiling information for performance tuning, and to guide hot-path and code layout compiler optimizations.

The rest of the paper has the following organization. Section 2 details an example motivating why cyclic behavior exists in applications. Section 3 describes Basic Block Distribution Analysis, and how it is used to find the end of initialization and the period length in applications. Section 4 examines the cyclic behavior of programs in terms of architectural features and metrics, in order to see how they correlate to the basic block cyclic behavior found using BBDA. Section 5 presents results using BBDA for finding places in an application to simulate. It also analyzes the error in using BBDA for finding the representative part of the program to simulate across a range of architecture features (IPC, branch prediction, value prediction, address prediction, cache designs, and reorder buffer occupancy). Section 6 describes related work, and section 7 summarizes the results and contributions of our work.

2 Cyclic Behavior of Programs

Most programs do not execute in a steady state, even at a high level. Instead they tend to go through different stages of execution, starting with a setup phase which is used to initialize data structures and set up for the rest of execution. This start-up time can account for a significant amount of execution. For example, the SPEC95 program wave needs to execute for almost 7 billion instructions before it reaches the code that accounts for the bulk of the execution.

Once the initialization stage has been past and we are in the bulk of the execution, there are still execution phases to be found. Programs tend to be written in a modular fashion, often as a set of procedures contained in a loop, where each procedure is then another loop with more procedures. While this mode of execution is not representative of every important program written, it is the common case for compute bound applications, the type that we concern ourselves with when examining new architectural modifications. Applications, when written in this manner, have a very strong *periodic* behavior, alternating between completely different sections of code.

If, as computer architects, we are not cognizant of the fact that programs execute in distinct phases, we may be testing the performance of our machine on a single very unrepresentative section of execution such as the initialization phase, or at the very least we may be over-representing parts of the program.

Figure 1 shows the behavior of wave as it executes. Plotted on the graph are a variety of architectural metrics such as IPC and cache miss rates. The graph shows that wave has very distinct phases of execution, starting with an initialization phase that ends around 7 billion instructions. After this, the program enters into a series of cycles, each made up of two phases. In one phase an average of over three IPC is achieved every cycle, while in the other phase the IPC drops down to under two. Complete details for this graph are discussed in Section 4.

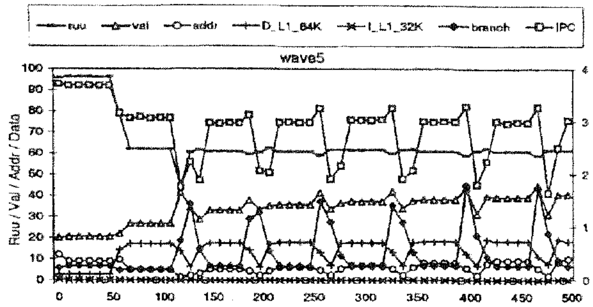


Figure 1: Time varying behavior for the SPEC95 program wave. Each unit of the X-axis represents 100 million committed instructions. Results are shown for IPC, reorder buffer (RUU) occupancy, value, address, and branch prediction miss rates, instruction and data cache miss rates using the Y-axis scale each metric is labelled upon.

The reason for this periodic behavior can be seen in the call graphs generated for wave shown in Figures 2 and 3. Figure 2 is the call graph generated for just the partial execution of wave during the sections of high IPC, while Figure 3 is the call graph for the sections of low IPC. The nodes on the graphs are procedures annotated with the number of times that they were called. The strong periodic behavior of wave is due to an outer function, not shown, calling two different routines in succession, `trans` and `field`. The call graphs show that `trans` and `field` do share some low level functions such as `__F_sqrt4` and `_OtsDivide32`, but the bulk of their execution occurs in different functions.

It is easy to see that if careful decisions are not made about where in a program's execution to simulate, we could easily see differences of a factor 2 in important metrics such as IPC. This example further demonstrates the fact that different phase behavior can be identified by examining the execution behavior of the code. This motivated us to develop a general automated technique for determining where to simulate based on the basic blocks of the program.

3 Basic Block Distribution Analysis

In this section we propose Basic Block Distribution Analysis as a generic way of determining the cyclic behavior of an application and finding preferred simulation points in the application in order to achieve a representative sample of its execution.

3.1 Basic Block Vectors

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies

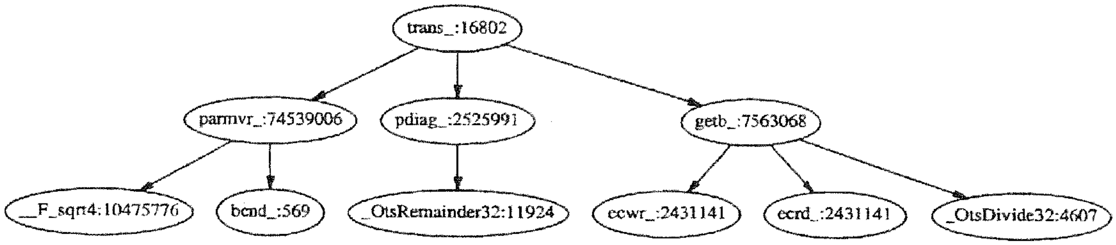


Figure 2: Call graph generated from wave for the phase of execution, where an average IPC of 3 is achieved.

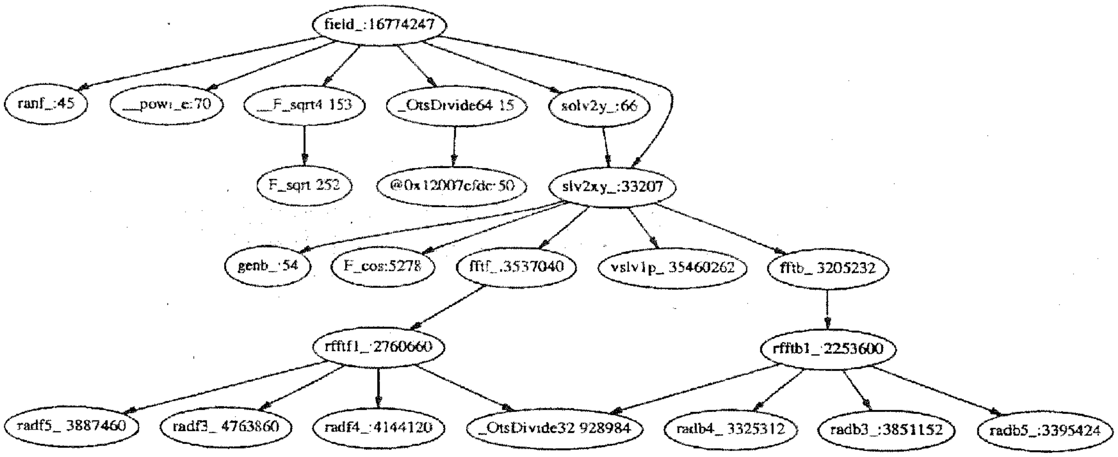


Figure 3: Call graph generated for the phase from wave, where there is an average IPC of 2.

with which basic blocks are executed as the metric to compare different sections of the application's execution. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing at that time, and basic blocks provide us with this information.

The program, when run for any interval of time, will execute each basic block in the program a certain number of times. Knowing this information provides us with a basic block *fingerprint* for that interval of execution, which tells where in the code the application is spending its time. The basic idea is to find a reasonable sized interval of time in the program's execution that has a basic block fingerprint similar to the full execution of the program. If we can find this, we know that both the full execution of the program and the interval we choose spends proportionally the same amount of time in the same code.

For the results in this paper, the basic block fingerprints are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, or *sample*, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next 100 million interval.

A *Basic Block Vector* (BBV) is a single dimensional array, where there is an element for each static basic block in the program. Each element in the array is the count of how many times a given basic block has been entered during an interval. It is sometimes useful to take Basic Block Vectors of varying size intervals. We say that a Basic Block Vector, which was gathered by counting basic block executions over an interval of N *times* 100 million instructions, is a *Basic Block Vector of duration N*.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the *proportions* of basic block execution, a BBV is normalized by having each element divided by the sum of all the elements in the vector. This normalization ensures that the sum of all the elements in the BBV is 1, which in turn allows us to compare vectors of different durations.

The Basic Block Vector that contains the normalized basic block frequencies for the entire execution of the program provides what we call the *target* BBV. It is the goal of the analysis that we present to find a Basic Block Vector, of small duration, that is very similar to the target BBV. In finding this we will have found a section of code that is representative of the whole.

In order to find a Basic Block Vector that is similar to the target BBV, we must first have some way of comparing two Basic Block Vectors. The operation we desire takes as input two Basic Block Vectors, and as output has a number which tells us how close they are to each other. To compute this function we take the element-wise subtraction of the two vectors. We then take the absolute value of each element, and sum all the elements together into a single number. This produces a number between 0 and 2, since each BBV sums to 1. We use this single number to tell us how closely related the two BBVs are. We call this the *difference* between the two BBVs. Now that we have a way of comparing two Basic Block Vectors, we can begin to look into how the execution of a program changes over time.

3.2 Creating a Basic Block Difference Graph

Before we can begin to understand how to find a representative interval of the program, we need to understand how the execution of a program changes over time. For this reason we create a Basic Block Difference Graph. The Basic Block Difference Graph is a plot of how well each individual sample in the program compares to the target Basic Block Vector created for the entire run to completion.

For each interval of 100 million instructions, we create a BBV of duration 1 and calculate its difference from the target BBV. Figure 4 shows the plot of all of the BBV differences across the entire execution creating a *Basic Block Difference Graph*. The x-axis is the number of instructions in 100 millions, and the y-axis is our measure of comparison between Basic Block Vectors discussed above, the BBV difference. A difference of 2 means that the two vectors are completely unrelated, while a deviation of 0 is the result of a perfect match between a BBV and the target vector.

In the following sections we describe how we use the basic block difference graph to (1) find the initialization phase of the program, and (2) find the period for that program.

3.3 Finding the End of Initialization Phase

Execution during the initialization phase of programs is very different from the steady state behavior of the application. In a study on value prediction [2], we found that `tomcatv` saw a 68% execution speedup using value prediction when simulating the initialization phase of the program, in comparison to 5.8% speedup after fast forwarding past the initialization

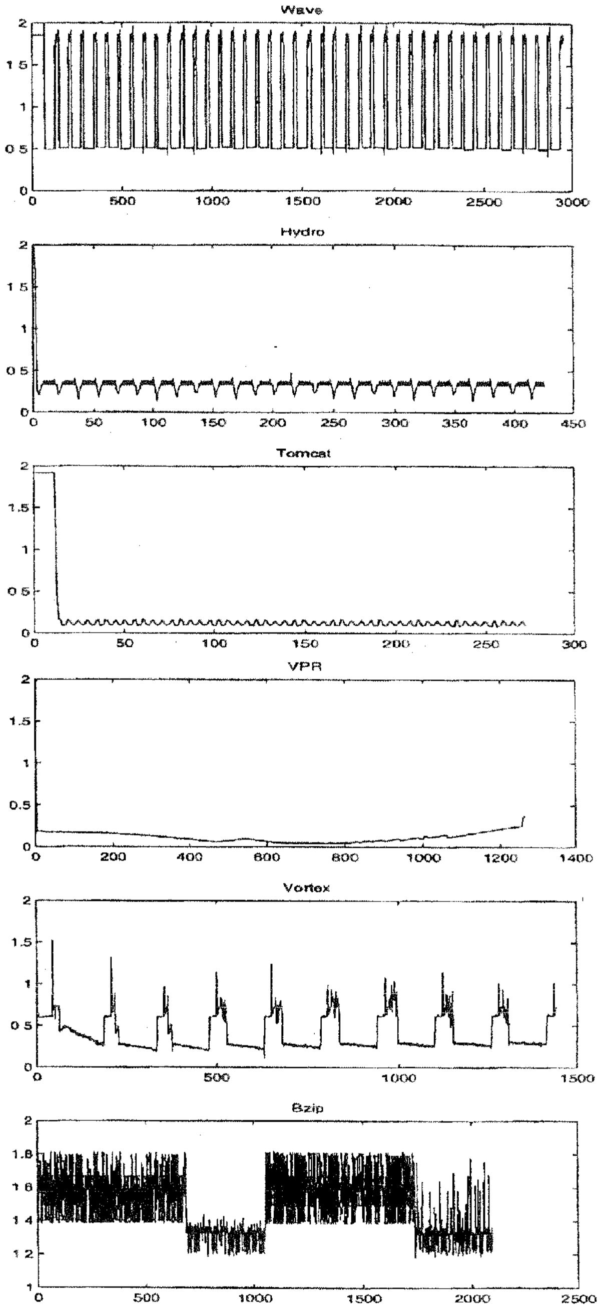


Figure 4: The basic block difference graphs. Each x-axis unit represents 100 million executed instructions. The graphs show the Basic Block Vector difference on the y-axis, which is calculated by comparing the target BBV with the BBV generated for each 100 million interval of executed instructions.

phase. In contrast, vortex saw an 11% execution speedup with value prediction in the initial part of the program, but saw a 27% execution speedup after fast forwarding. These results show that results generated for only the beginning of execution can be terribly misleading, and that it is very important to simulate representative sections of code.

The approach we use to determine the end of the initialization stage can be thought of as sliding a piece of jig-saw puzzle over the rest of the puzzle. Since the jig-saw piece will fit best at the spot it is removed from, the comparison at that point will show the least difference. However, as soon as it is shifted away from its space, the comparison with the underlying pieces will show a marked difference.

To find the end of the initialization phase we treat the BB difference graph as a signal. We take the first quarter of the BB difference graph (signal), which we call the *Initialization Representative Signal* (IRS), and we use this to search for the end of the initialization. We take the IRS and slide it over the BB difference signal looking for the first peak where the IRS differs from the BB difference signal. In this way we treat the BB difference graph as the puzzle, and the IRS as the piece of the puzzle we are sliding across.

We chose IRS to be the first quarter of the BB difference signal to capture the majority if not all of the initialization stage. This is based on the assumption that the initialization phase will be shorter than half the length of the entire execution.

We compare the IRS at every point across the first half of the original BB difference signal. A signal starting at each point in the BB difference graph equal in length to the IRS signal, is compared to the IRS. To compare these two sub-signals we take the absolute difference of each point of the two sub-signals, and summarize the resulting differences into a single number. This number represents how close these two signals match up. This is done for every point within the first half of execution in the BB difference graph resulting in a new graph, which we call the *Initialization Difference Graph*. These are shown in Figure 5.

The graphs can be categorized into two observable behaviors. A periodic pattern as seen with wave, vortex, and bzip, is due to the IRS containing the initialization stage as well as some cyclical behavior from the execution. This is enough to manifest the cyclical behavior during the remainder of the comparison past the initialization stage. A steep incline with a plateau is seen with hydro, tomcat and vpr. The plateau is explained by the initialization part of these programs not having any overlap with the rest of the program after the initialization phase is completed.

From the programs we examined, the initialization stage is complete at the first peak or corner in the initialization difference graph. When the initialization representative signal finally reaches the end of the initialization stage on the BB difference signal, the difference is maximized since there is no more of the initialization phase left to compare to.

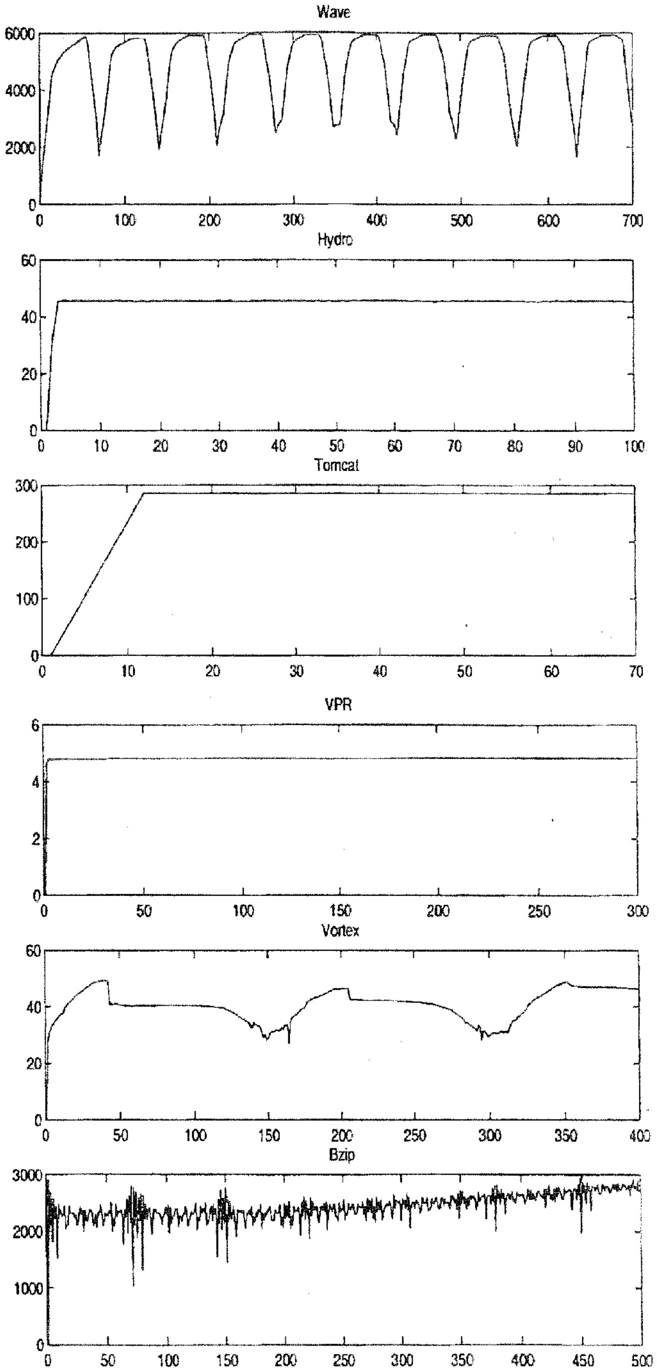


Figure 5: The initialization difference graphs. The x-axis units are in terms of 100 million instructions. The y-axis for each x-axis value represents the signal difference between the IRS and the original basic block difference signal *starting* at that x-axis value.

Mathematically, a peak or a corner in a graph represents the point where the slope is changing the fastest. The second derivative is a function of the rate of change of the slope, and is used in our algorithm to determine this point marking the end of the initialization. The first column in Table 2 shows the end of initialization points that are automatically found using the above analysis.

3.4 Finding the Period

To find the period we form a *Period Representative Signal* (PRS) from the BB difference graph starting at the precomputed end of the initialization phase found in the previous section. The PRS we use is one quarter the length of the program's execution. We found that duration to be sufficient to capture periods of length (duration) comprising up to half of the program's execution.

To find the period we slide the PRS across half the entire BB difference graph, starting at the end of the initialization stage. We perform the same comparisons for each x-axis value as above for finding the initialization stage, resulting in *Period Difference Graphs* shown in Figure 6.

The period graph shows all of the points where the PRS matched the sub-signals from the original signal (BB difference graph). After shifting the PRS over the BB difference graph, the resulting calculations close to zero represent a match of the PRS to the original sub-signal. The time duration between each match represents the period for the program. Therefore, all of the local minimums from shifting the PRS are used to calculate the period. The period is calculated by taking these minimum Y-axis points in the period graph, and calculating the length in instructions (X-axis) between these minimums. This length is the period of the signal, and the period of the application. The second column in Table 2 shows the periods that are automatically found using the above analysis.

The two programs that do not fit cleanly into our description for finding periodic behavior are `vpr` and `bzip`. `Vpr` does not have a very visible period, and its behavior is not very repetitive. However, we still find very good representative points for simulation for `vpr` as is shown using the analysis we present in section 5.

`Bzip` on the other hand has multiple periods. The first and largest period has a duration of 1046 as seen in Figure 4, which consists of 2 cycles over the complete execution of `bzip`. In looking at Figure 7, we can see how the behavior is captured when creating a BB difference graph using different BBV durations. Results are shown for using basic block vectors with duration of 6, 12, and 52 (hundred of million instructions) to create the BB difference graph. For a vector duration of 6, we find that the next period to be found has a duration of 78, and the smallest period is of size 9. These figures also show that using larger durations of a BBV creates a BB difference graph that emphasizes the larger periods.

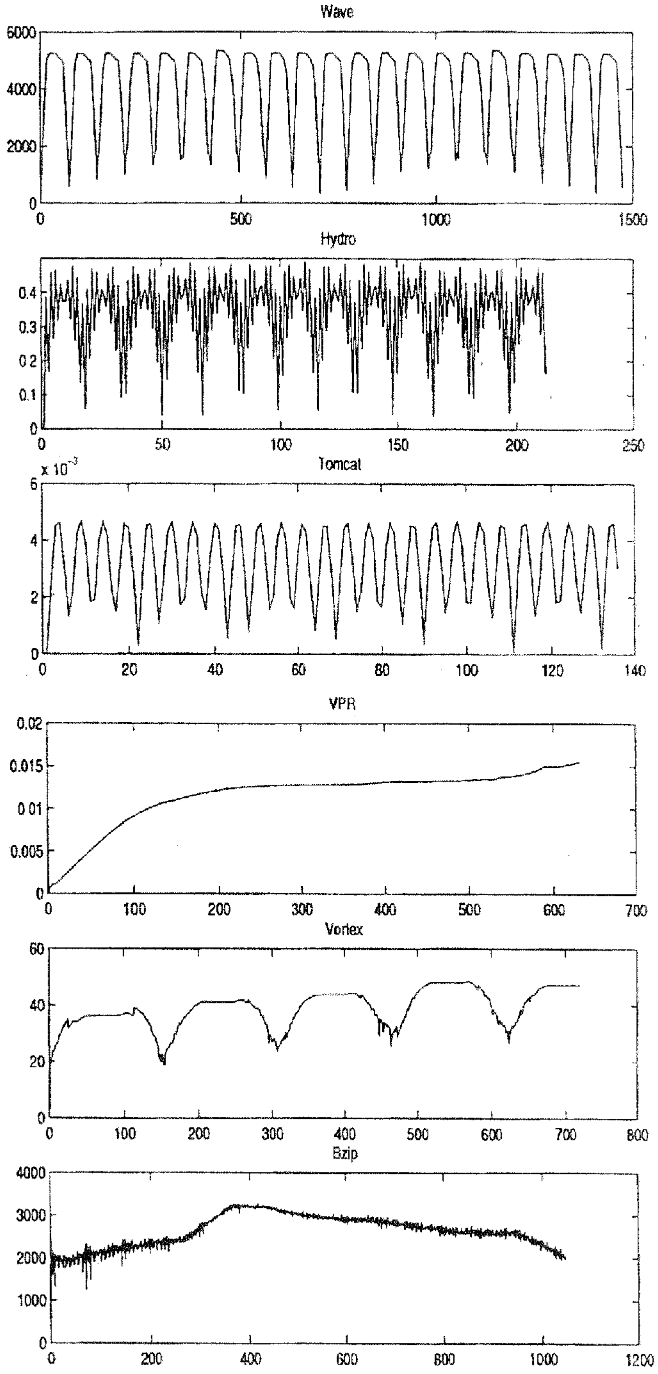


Figure 6: The period difference graphs. The x-axis units are in terms of 100 million instructions. The y-axis for each x-axis point represents the signal difference between the PRS and the original basic block difference signal *starting* after the end of the initialization phase.

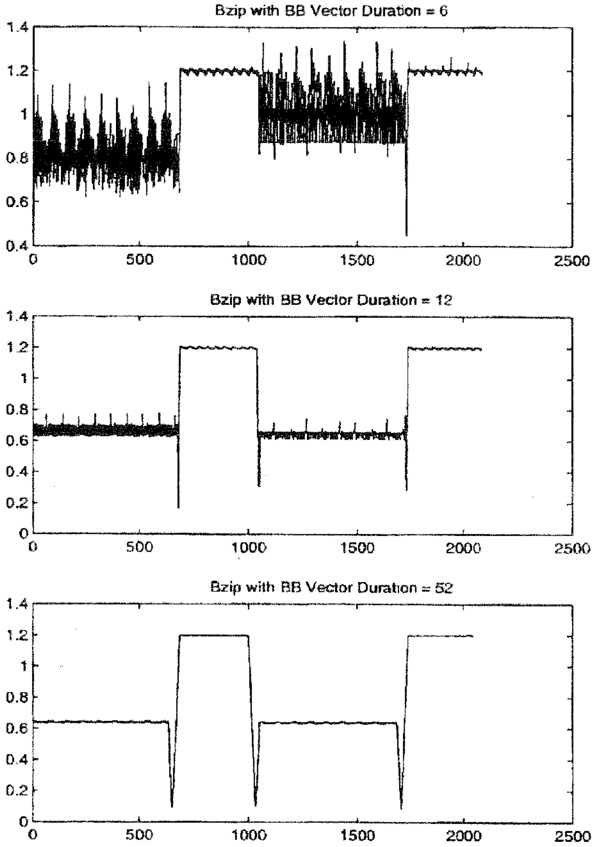


Figure 7: The basic block difference graphs for `bzip` varying duration of the basic block vector when creating the graphs. The results show that using a larger BB vector creates a BB difference graph that emphasizes the larger periods.

3.5 Fourier Analysis

We initially tried to use Fourier analysis to discover the period of a signal. We convolved the signal with itself to smooth out the basic block difference graph, emphasizing the frequencies with larger amplitude. The convolution accentuates the periodic behavior of the original signal, but this new signal still had to be analyzed to find the period. For `vortex`, the new signal was actually inferior to the original, since `vortex` has a slightly varying period throughout its entire execution. The convolution did not work well for signals that did not have static period lengths.

The Fourier analysis potentially could have benefits when dealing with certain types of execution. Our period algorithm currently computes a single period for the entire execution. This is not always the optimal period, because there could locally be periodic behavior throughout the execution. `Bzip` is a prime example of this as described above. `Bzip` has two distinct phases, and each phase has its own periodic behavior but the signal is very noisy. Fourier analysis could potentially provide information about all the periodic behavior in a signal, and extracting this to optimize our current approach is part of future work.

Instruction Cache	32k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	64k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 12 cycle latency
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

4 Cyclic Behavior of Architectural Metrics

In this section we examine the time varying behavior of applications in terms of architectural features and metrics. We show the correlation between the periodic behavior found via BBDA and the architectural features and metrics examined during simulation.

4.1 Methodology

To examine these architecture features and metrics, we collected information for three of SPEC95 programs (`tomcatv`, `hydro`, and `wave`) and three of the SPEC 2000 programs (`bzip`, `vortex`, `vpr`) for their reference input sets. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, C++, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used was derived from the SimpleScalar 3.0a tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. The simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction. The baseline microarchitecture model is detailed in Table 1. We modified the 3.0a release of SimpleScalar, so that the memory hierarchy buses were pipelined, with a transfer width of 8 bytes per cycle.

4.2 Collecting Time Varying Behavior

To show the time varying behavior of the programs, SimpleScalar was modified to output and clear its statistics after every 100 million *committed* instructions. Only the statistic counters are cleared between intervals, and the state of the machine (e.g., cache and branch prediction tables) are *not* cleared between intervals. This eliminates any cold-start error from being added into the experiment.

Results are then graphed for every 100 million committed instructions for the programs examined. This should yield a clear picture of the large scale runtime behavior exhibited by each application as well as indicating which sets of instructions are more indicative of the execution as a whole. It is, however, of small enough granularity that it provides useful information about program start up times and can be easily simulated on any machine. Each program was run until completion, but we only graph enough intervals to show the cyclic nature for each program.

The following summarizes the data graphed:

- **Instructions Per Cycle.** This is the number of instructions that are committed in each sample, which is always 100 million, divided by the number of simulated cycles that it took to execute those instructions.

- **Percent RUU Occupancy.** SimpleScalar uses a unified Register Update Unit (RUU) to model its reorder buffer and reservation stations [12]. In our simulations we used a 128 entry RUU, and report results in terms of the percent of the RUU entries used on average during a 100 million instruction sampling period.

- **Cache Miss Rate.** Cache miss rates are shown for a 32 KB 2-way associative instruction cache, and a 64 KB 4-way associative data cache. Both caches have 32 byte lines.

- **Branch Prediction Miss Rate.** We used McFarling's bi-modal gshare branch predictor [7]. An 8K entry 2-bit chooser table is used to choose between an 8K entry 2-bit bi-modal branch predictor and an 8K entry gshare table. A 256 entry 4-way associative branch target buffer is used to provide the predicted addresses, and a 32 entry return address stack is used to predict return instructions. The branch misprediction rate over all the types of executed branch instructions is shown.

- **Address Prediction Miss Rate.** Miss rates are shown for 2-delta stride address prediction for an infinite sized table (each load gets its own entry) [5, 9]. The 2-delta address predictor will only change it's prediction if the stride is seen two times in a row. Miss rates are shown for only applying address prediction to load instructions.

- **Value Prediction Miss Rate.** Miss rates are shown for 2-delta value and address prediction for an infinite sized table [4, 13]. The 2-delta value predictor will only change the stride if seen two times in a row. Miss rates are shown for only applying value prediction to load instructions.

Note, address and value prediction were not used for architectural optimizations in gathering these results, only their miss rates were gathered. Therefore, they do not affect the IPC, branch or cache miss rate results being shown.

4.2.1 Cyclic Architecture Results

Figure 8 and Figure 1 show the time varying behavior of the 6 SPEC programs we examined. The legend is at the top of each figure. For each program, the results for IPC, average percent RUU occupancy, percent branch miss rate, percent value miss rate, percent address miss rate, and percent instruction and data cache miss rates are shown on the same graph. Since all of these different results are shown on the same graph, each graph has two Y-axis.

For each graph, the left and right Y-axis are labeled with the metrics that use that axis. For most of the graphs, percent RUU occupancy, and value and address miss rates use the left Y-axis. Similarly, I-Cache miss rate, branch miss rate, and IPC usually use the right Y-axis. The D-Cache miss rate is shown on either axis depending upon the program and axis scale in order to allow interesting trends to be seen.

The X-axis is in terms of 100 million *committed* instructions. We ran all of the programs to completion, and found them to either (1) converge to a constant behavior until the last few 100 million instructions, or (2) have a repeatable cyclic behavior until the end of their execution. Because of this, and to save space, we only show enough of the program to demonstrate the cycles we found. For `hydro`, `tomcat`, and `bzip`, 5 billion instructions is enough to clearly demonstrate the cyclic nature of the programs. `Vortex` has cycles of a much larger scale, on the order of 150 billion instructions, and `wave` has cycles on the order of 7 billion instructions. `Vpr` has mild cyclic tendencies, but the pattern is not as concrete as for other programs.

5 Choosing Where to Simulate and Error Analysis

SimpleScalar [1], one of the fastest simulators, executes on the order of 1000 times slower than hardware. SimpleScalar emulates the execution of a program and allows the simulation to execute down speculative paths of execution. This is critical for accurately modeling speculative execution and recovery techniques for many of the latest architecture features being studied in the field. Most researchers use a cycle level simulator similar to SimpleScalar, executing only

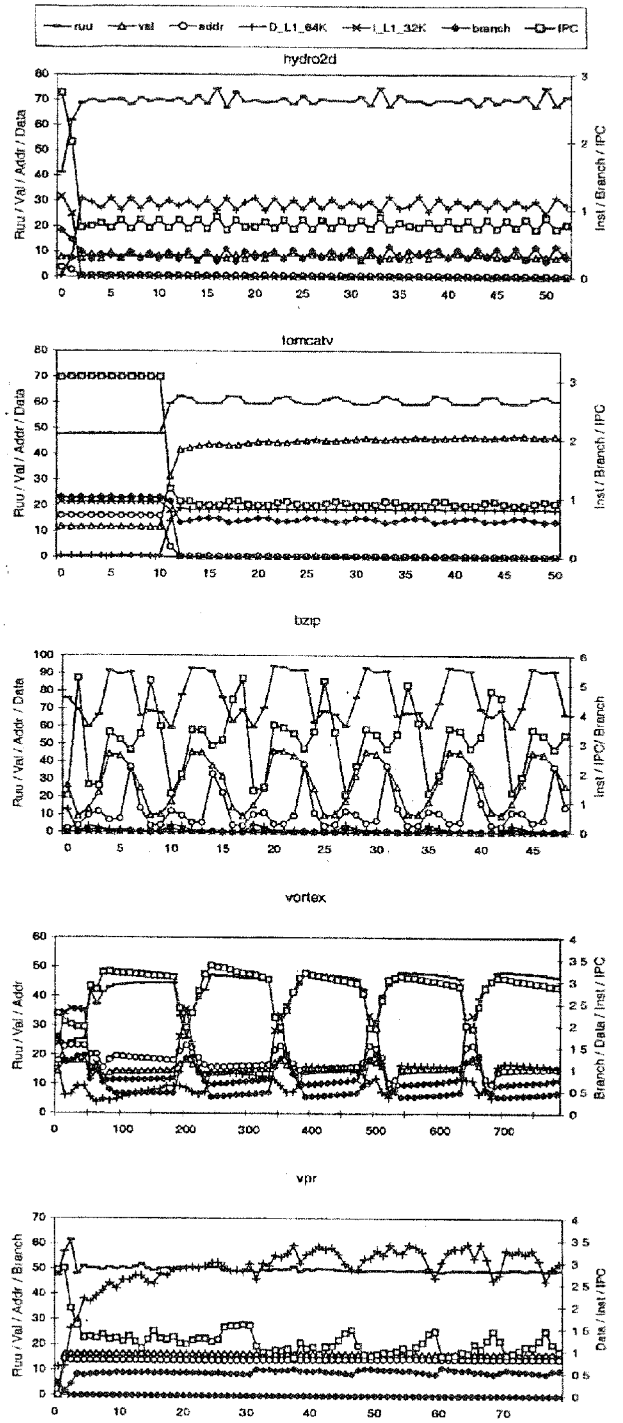


Figure 8: Time varying behavior for the programs hydro2d, tomcatv, bzip, vortex, and vpr. Each unit of the X-axis represents 100 million committed instructions.

a small fraction of the program. A few hundred million instructions may be typically executed, starting from a predetermined point.

In Table 2 we see the baseline behavior of the six programs for the architectural metrics discussed in section 4 for the complete run of the application. In addition to this we have included the initialization phase and the period duration (in 100 of millions of instructions) as determined by using the BBDA analysis discussed in section 3. We have found that to get the most representative sample of a program, at least one full period must be simulated.

5.1 Simulation Points

To evaluate the accuracy of the period length found using BBDA, we now compare the behavior of simulating for a single period to that of simulating the program's complete execution. We choose a preferred period as the simulation starting point by building a BB difference graph for each program with a BBV duration equal to the period length shown in Table 2. We then take the minimum point in this new BB difference graph as the the preferred period to simulate.

Table 3 compares the performance of several different metrics for the preferred period simulated and compares this with the baseline metrics shown in Table 2. The column labeled start is where the simulation was started from, and the simulation was ran for one complete period with the length shown in Table 2. For each of these experiments, cold start effects were eliminated by warming up the simulator with the full execution of the program to that point. The metrics examined are the same as examined in Section 4. In addition to this, associated with each metric is an error. The error is the percent difference between the metric measured over the preferred period we simulated versus the complete execution of the program.

The IPC values for the periods match very closely with the execution of the program as a whole. For all the programs there was less than a 5% difference between the IPC of the preferred simulation period and the full program execution. Most of the other metrics match up very closely as well. We show '–' for instruction cache error results for most of the programs, since the instruction cache miss rates were too low (below 0.05%) to represent any meaningful error.

The results for vpr show that we were able to capture its IPC within 4.3% when simulating 200 million instructions (one period) starting 74.6 billion instructions into the program. Even though there are different basic blocks executing in different proportions across the run of the application, the chosen sample is still very close to the execution as a whole.

5.2 Limited Simulation

Due to time constraints a researcher cannot typically simulate the whole program, but instead can simulate only for a few hundred million instructions, which is usually smaller

name	init	period	bpred	ruu	IPC	d miss	i miss	val miss	addr miss
bzip	2	9	4.2%	75.8%	2.681	1.7%	0.000%	25.1%	13.3%
hydro	5	17	0.4%	68.7%	0.793	14.6%	0.022%	8.3%	0 6%
tomcat	13	5	0.8%	59.6%	0.955	9.7%	0.043%	46.2%	1.0%
vortex	40	144	0.6%	43.4%	2.726	0.9%	0.979%	15.2%	16.4%
vpr	4	2	9.3%	49.8%	1.143	3.0%	0.001%	16.6%	14.2%
wave	68	70	0.6%	62.2%	2.596	7.4%	0.000%	38.1%	7.9%

Table 2: The first two columns show the length of the initialization phase and the size of the period in hundreds of millions of instructions. The average branch misprediction rate, ruu occupancy, instructions per cycle, data cache miss rate, instruction cache miss rate, value misprediction rate, and address miss prediction rate are also shown for the full run to completion.

than the period. To determine where to simulate given this constraint, we build a BB difference graph for each program with a BB vector duration of N, where N is the number of instructions in 100 of millions the user is willing to simulate. We then take the minimum point of that graph to represent the ideal simulation point.

Table 4 shows the effect of using only a limited amount of simulation time. Here we limit the amount of simulation time to only 300 million committed instructions, starting at the instruction, in 100 of millions, shown in the first column of Table 4. We can see that the error rate has gone up over that in Table 3. However, due to the fact that we have carefully selected our starting point with the algorithms presented in Section 3, the results we get are within acceptable bounds. The worst case IPC difference is 6%

The one program that does not do well with the smaller run size is `bzip`. For `bzip` the address miss rate and the value miss rate are off by around 80%. As our periodic results show, 900 million simulated instructions are needed to capture the small period in `bzip`, and simulating for 300 million instructions was simply too small to capture the behavior of the loop.

We now examine the performance of choosing our simulation point by picking it to be just after the initialization phase. Table 5 shows the same metrics as presented in Table 4 for a section of execution past the initialization stage by one period. The start of simulation is chosen to be the initialization time plus the time for one period. The intuition behind this is to simulate the soonest time past initialization, while still allowing for a full period of simulation to "warm up" the architectural structures such as the cache and branch predictor. In looking at Table 5, we see that using this scheme provides higher errors for important metrics such as IPC, branch prediction and data cache miss rates over using BBDA to find a preferred starting point as shown in Table 4.

6 Related Work

In this section we describe work related to finding simulation points, techniques for using sampling for simulation, and statistical simulation.

6.1 Time Varying Behavior of Programs

We presented in [10] a first attempt at showing the periodic patterns and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy. Skadron et. al [11] also examined creating similar time varying graphs for branch miss rates. They then used these graphs to manually choose where to fast-forward to pick out the simulation periods, similar as to what we proposed in [10].

6.2 Automatically Finding Where to Simulate

Concurrent to the work presented in this paper, Lafage and Seznec proposed an automated approach for choosing representative slices of a program's execution [6].

They propose a technique similar to [10] to gather statistics over the execution of the program to completion. There are two major differences. First, they propose to use metrics that are architecture independent to characterize the behavior of the program. They evaluate two such metrics, one which captures spatial locality and one which captures temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of the both the program's full execution and the execution of samples. The second point they propose is to use clustering and choosing algorithms to find a set of samples which captures the full execution of the program.

Our approach is cooperative in that the metrics and analysis we propose, and the clustering and choosing algorithms developed in [6] could be easily used together, and this is an area of future research.

6.3 Statistical Sampling

Our basic block distribution analysis accurately finds represetive periods for simulation, but some of these periods are still too long for conducting detailed simulation studies. Therefore, in section 5 we examine choosing a few hundred million instructions to simulate from these long periods, and

name	start	bpred	err	ruu	err	IPC	err	data	err	inst	err	val	err	addr	err
bzip	150	4.2%	1%	75.4%	0.5%	2.8	5.1%	1.3%	25.8%	0.0%	–	25.4%	1.1%	15.7%	17.9%
hydro	6	0.3%	16%	69.8%	1.7%	0.8	2.5%	14.8%	1.5%	0.0%	–	8.2%	1.8%	0.6%	9.1%
tomcat	12	0.8%	3%	60.5%	1.5%	0.9	1.5%	9.8%	1.1%	0.0%	–	41.1%	12.4%	0.9%	17.1%
vortex	382	0.6%	2%	43.7%	0.8%	2.8	1.9%	0.9%	1.2%	1.0%	2.8%	15.2%	0.1%	16.3%	0.7%
vpr	746	9.0%	3%	49.7%	0.3%	1.2	4.3%	3.1%	6.4%	0.0%	–	16.6%	0.0%	14.4%	1.3%
wave	127	0.6%	9%	60.7%	2.5%	2.5	3.3%	7.7%	4.4%	0.0%	–	40.4%	6.1%	8.5%	7.8%

Table 3: Results for simulating one complete period through the application. The first column shows the starting instruction of the period simulated (in 100s of millions of instructions). The branch missprediction rate, ruu occupancy, instructions per cycle, data cache miss rate, instruction cache miss rate, value misprediction rate, and address miss prediction rate are shown for one complete cycle of the program's execution. Next to each of these columns is the percent difference between that metric for the period and the same metric for the full program execution.

name	start	bpred	err	ruu	err	IPC	err	data	err	inst	err	val	err	addr	err
bzip	1733	4.0%	6%	63.8%	18.8%	2.5	5.9%	1.8%	8.0%	0.0%	–	14.2%	77.2%	7.3%	82.0%
hydro	36	0.3%	12%	69.2%	0.9%	0.8	3.9%	14.8%	1.4%	0.0%	–	8.4%	0.4%	0.6%	9.3%
tomcat	144	0.8%	1%	60.9%	2.2%	1.0	1.9%	9.5%	2.0%	0.1%	–	39.8%	16.2%	1.1%	13.7%
vortex	330	0.6%	3%	41.9%	3.6%	2.8	3.4%	0.7%	16.3%	1.0%	4.0%	15.7%	3.8%	17.7%	7.7%
vpr	746	9.0%	3%	49.7%	0.3%	1.2	4.3%	3.1%	6.4%	0.0%	–	16.6%	0.0%	14.4%	1.3%
wave	1036	0.3%	84%	61.5%	1.2%	2.8	6.0%	7.9%	6.7%	0.0%	–	37.0%	2.8%	6.5%	20.8%

Table 4: The same metrics as presented in Table 3, but for an automatically chosen 300 million instruction simulation point. The error from comparing the sampled metric to the full execution of the program is listed next to each metric.

name	start	bpred	err	ruu	err	IPC	err	data	err	inst	err	val	err	addr	err
bzip	11	4.9%	17%	74.3%	2.0%	2.2	23.2%	2.8%	68.9%	0.0%	–	22.7%	10.8%	8.5%	55.4%
hydro	22	0.3%	12%	69.6%	1.4%	0.8	2.4%	14.8%	1.7%	0.0%	–	8.5%	1.8%	0.6%	8.6%
tomcat	18	0.6%	28%	61.0%	2.4%	0.9	4.6%	10.1%	5.1%	0.0%	–	44.0%	6.1%	0.3%	237%
vortex	184	0.4%	42%	46.4%	6.9%	3.2	17.6%	0.9%	6.1%	0.7%	36%	14.8%	2.3%	16.2%	1.6%
vpr	6	1.1%	740%	58.1%	16.6%	3.0	162%	0.4%	621%	0.0%	–	16.6%	0.2%	13.8%	2.6%
wave	138	0.9%	55%	60.5%	2.8%	2.4	9.1%	7.3%	1.1%	0.0%	–	40.1%	5.5%	7.7%	2.3%

Table 5: The same metrics as presented in Table 3 for a section of 300 million simulated instructions chosen one period after the end of the initialization phase. Next to each of these columns is the percent difference between that metric for the chosen simulation and the same metric for the full program execution.

we showed how close this approach comes to the overall execution of the applications we examined. Another approach is to use sampling simulation inside of a representative period found using BDDA in order to maintain accuracy while reducing simulation time.

Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These techniques take a number of contiguous execution samples, referred to as clusters in [3], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated.

To use sampling one has to address the issue of how to deal with the state of the machine when switching from one cluster to starting the simulation of another cluster. One option for providing meaningful results is to first sample a large number sequential instructions in order to provide results due to the time it takes to warm up the architecture structures (.e.g, caches) as well as taking a large number of samples to be sure to capture the large scale behavior of the program. Conte et.al. [3] proposed another option for the reconciliation of such disjoint sample points, whereby the structures holding state are not reset between clusters. For example, the cache would not be flushed and the BTB would not be reset when switching simulation from one cluster to the next. The hope is that the state of the machine from the end of one cluster is similar to the start of another disjoint cluster.

6.4 Statistical Simulation

Another technique to improve simulation time is to use statistical simulation such as that presented by Oskin et al. [8]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. This technique could also benefit from Basic Block Distribution Analysis. First, by using BBDA there may not be a need to execute the programs to completion in the first place, a very time consuming step. Second, separate traces could be gathered for different phases, rather than trying to get one phase that represents the average behavior of application as a whole.

7 Summary

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program.

In this paper, we present Basic Block Distribution Analysis as an automated approach for finding where to simulate in order to achieve an accurate estimate of the complete program. The basic block distribution of the program's entire execution can be gathered quickly and efficiently using a basic block or edge profiler, with no need for cycle accurate simulation. The basic block distribution we form from this profile acts as a fingerprint for the whole program's behavior. This fingerprint is then used to automatically find the end of the initialization phase and the period duration for the programs we examine. We then quantify and show that basic block distribution analysis is highly correlated with architectural metrics including IPC, branch miss rate, cache miss rates, value misprediction, address misprediction, and reorder buffer occupancy.

Our results show that if we simulate the application for one complete period that the IPC error rates are 5% or less for the programs we examine. We further show that if we are constrained to only 300 million instructions of simulation time that the most representative instructions are not necessarily found right after the initialization phase, but rather typically straddle the transition from one phase to the next. Using basic block distribution analysis, we show that it is possible to find these small representative sections of the program, which result in an error in IPC of 6% or less.

Acknowledgments

We would like to thank the anonymous reviewers for providing useful comments on this paper. This work was funded in part by DARPA/ITO under contract number DABT63-98-C-0045, and a grant from Compaq Computer Corporation.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction Level Parallelism*, May 2000.

[3] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[4] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[5] J. Gonzalez and A. Gonzalez. Memory address prediction for data speculation. Technical report, Universitat Politecnica de Catalunya, 1996.

[6] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[7] S. McFarling and J. Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. Association for Computing Machinery, 1986.

[8] M. Oskin, F. T. Chong, and M. Farrens. Hls: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[9] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, 1998.

[10] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[11] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–81, November 1999.

[12] G.S. Sohi. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[13] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.